Object-Oriented Analysis and Modeling

Course Overview

Objectives

❐   To make students familiar with practices of analysts in software-intensive projects.

❐   To provide a comprehensive explanation of UML.

❐   To explain advanced modeling concepts.

❐   To train the usage of UML by solving (almost) real-world problems.

Program

- Introduction.

- Generic modeling mechanisms.

- Modeling requirements and use cases.

- Modeling classes.

- Modeling composite structures.

- Modeling interactions.

- Modeling state machines.

- Modeling activities.

- Modeling components.

- Modeling deployment.

- Auxiliary constructs.

- Extensibility mechanisms and UML profiles.

- Advanced modeling mechanisms.

References

- *Unified Modeling Language: Superstructure.* Version 2.1.1, formal/2007-02-05, OMG, February 2007.

- M. Fowler: *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 3rd Edition, Addison-Wesley, September 2003.

- G. Booch, J. Rumbaugh, and I. Jacobson: *Unified Modeling Language User Guide.* 2nd ed., Addison-Wesley, May 2005.

- M. J. Chonoles and J. A. Schardt: *UML 2 for Dummies.* Wiley, July 2003.

- D. Pilone and N. Pitman: *UML 2.0 in a Nutshell.* 2nd ed., O'Reilly, June 2005.

- T. A. Pender: *UML Weekend Crash Course.* Wiley, April 2002.

- P. Kimmel: *UML Demystified.* McGraw-Hill Osborne Media, October 2005.

- D. Pilone: *UML 2.0 Pocket Reference.* O'Reilly, March 2006.

http://www.dcs.fmph.uniba.sk/~cervenka/ooam

Object-Oriented Analysis and Modeling

Introduction

Software Development

The value of software

❑   A crucial part of many industries.

❑   Expanding of size, complexity, distribution and importance.

Development limits of software industry

❑   It is very difficult to repeatable build and maintain large, complex, distributed and critical software systems with a good quality.

❑   Still quite a young industry.

❑   Software engineering is not developed as much as other engineering areas.

Symptoms of Software Development Failures

❒ Inaccurate understanding of end-users needs.

❒ Inability to deal with changing requirements.

❒ Modules that don't fit together.

❒ Software that's hard to maintain or extend.

❒ Late discovery of serious project flaws.

❒ Poor software quality.

❒ Unacceptable software's performance.

❒ Team members in each other's way, making it impossible to reconstruct who changed what, when, and why.

❒ An untrustworthy build-and-release process.

…

Waterfall Software Life Cycle
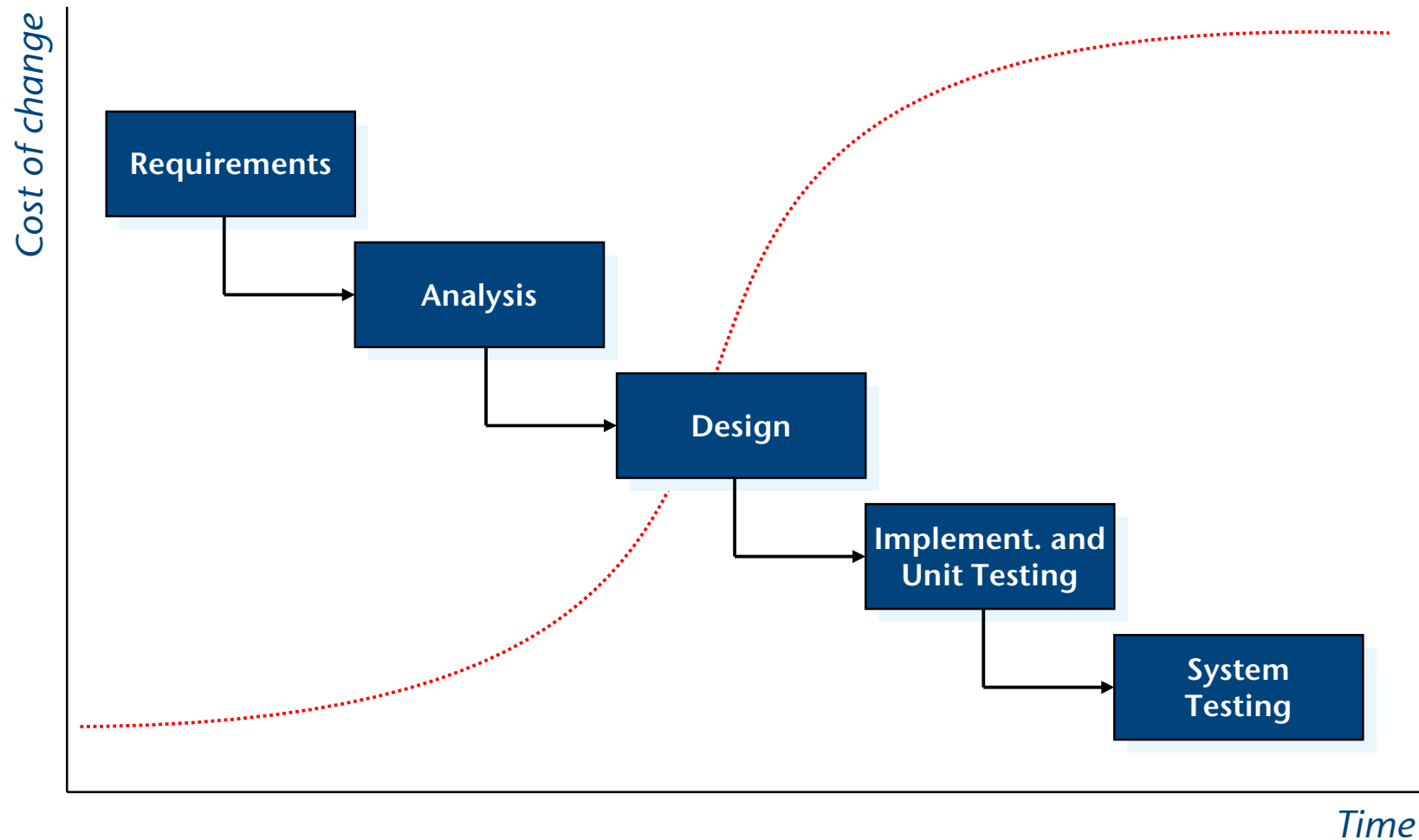
# Iterative-incremental Software Life Cycle
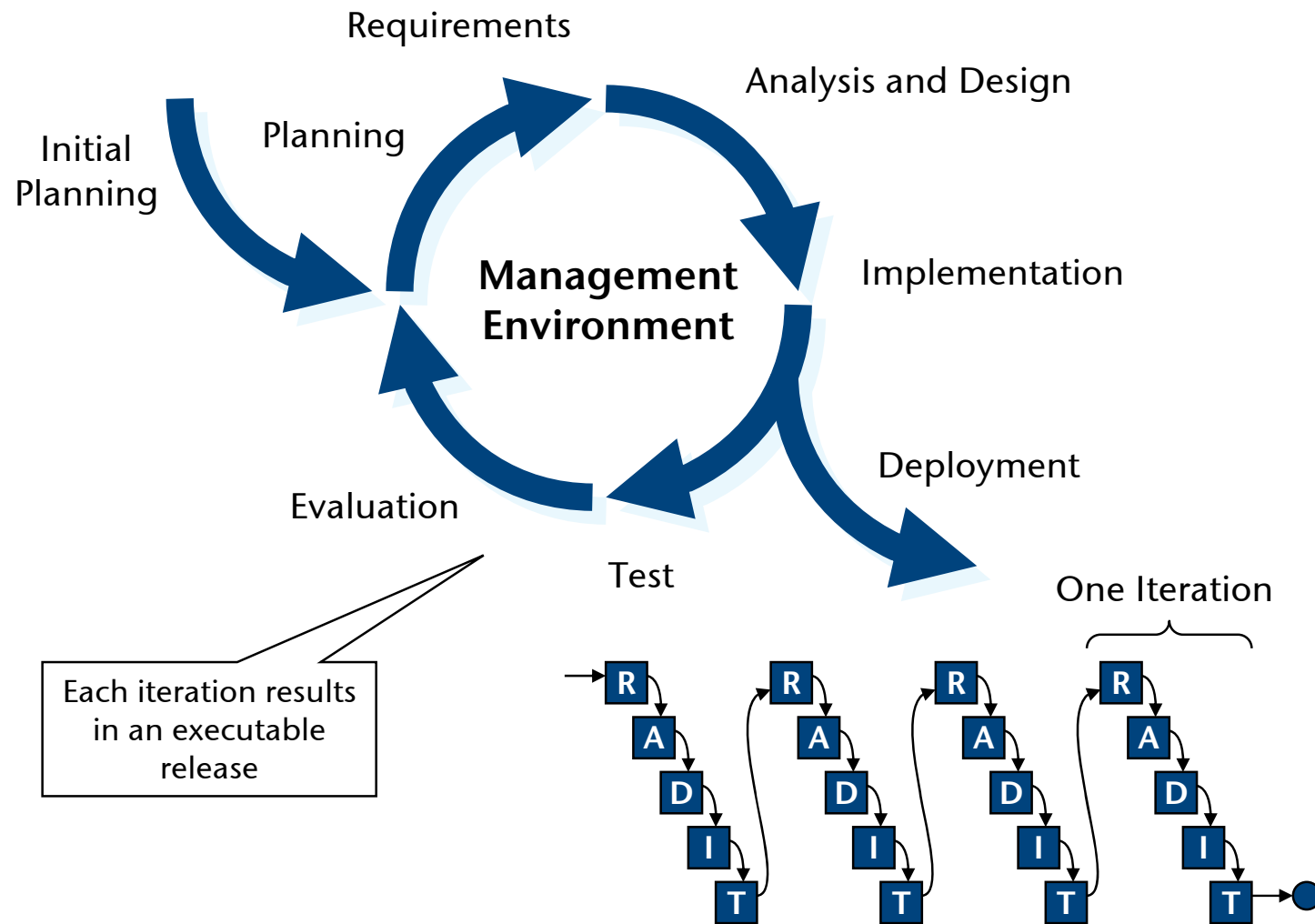
# Best Practices of Software Engineering

- ❏ Incremental change and iteration of solution.

- ❏ Short release cycles and iterations; "driving the car".

- ❏ Small initial investments.

- ❏ Manage requirements.

- ❏ A simple design.

- ❏ Lots of practice in modifying the design.

- ❏ Control changes to software.

- ❏ Visually model software.

- ❏ Verify software quality.

- ❏ Good communication.

- ❏ Rapid feedback.

- ❏ Courage.

…

Cost of change vs *Time*
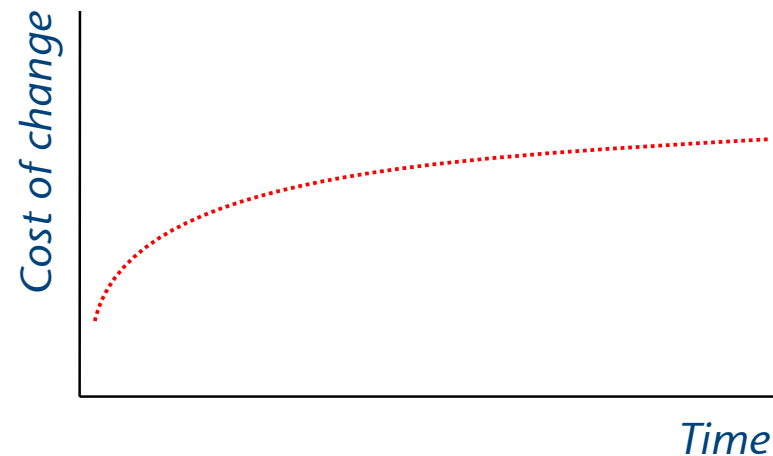
Process Example: Rational Unified Process

Visual Modeling
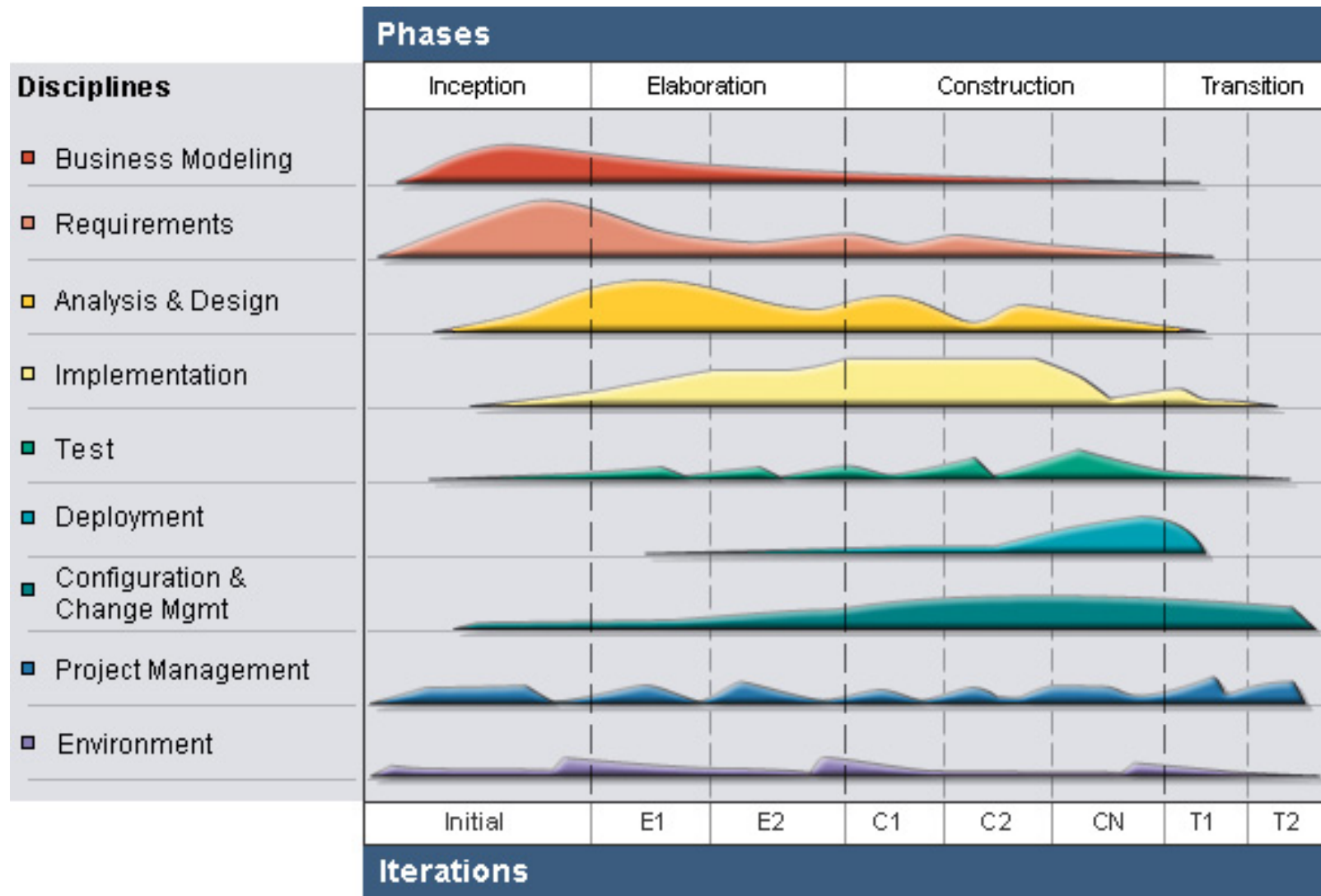
"One picture is better than a thousand words."

❒ Abstraction and simplification of reality.

❒ Different perspectives.

❒ Dealing with complexity (software is inherently complex).

❒ Better structuring of software architecture.

❒ Easy to understand, communicate and modify.

CASE Tools

Computer-Aided Software Engineering

❏ Tools supporting activities of the SW lifecycle:

- requirements management,

- analysis,

- design,

- code generation,

- reverse and round-trip engineering,

- documentation,

- configuration management,

...

❏ Usage of visual modeling.

→ **Shorter application development time with increased quality.**

Object-Oriented Analysis and Modeling

Introduction to UML

UML: Unified Modeling Language

The ***Unified Modeling Language*** (***UML***) is a language for *specifying*, *constructing*, *visualizing* and *documenting* the artifacts of a software-intensive system.

❑ Can be used also for other domains, such as, business modeling, process engineering, and others non-software systems.

❑ Stems out from modeling languages: Booch, OMT, OOSE, and others.

❑ Original authors: Grady Booch, Jim Rumbaugh and Ivar Jacobson (all from Rational Software Corp.)

❑ Now: UML is an OMG standard with several contributors.

History

Fragmentation:

- 1991: Booch '91, OMT-1, OOSE, ...

- 1993: Booch '93, OMT-2

Unification

- 1995: Unified Method (OOPSLA '95)

- 1996: UML 0.9 and UML 0.91

Standardization and industrialization

- 1997: UML 1.0 and UML 1.1

- 1998-2003:several minor revisions in UML 1.3, 1.4, and 1.5

- 2005: ISO/IEC 19501:2005 Information technology–Open Distributed Processing–Unified Modeling Language (UML) Version 1.4.2

- 2003-2005: UML 2.0

- 2006: UML2.1 (not formal)

- 2007: UML 2.1.1

Goals

❐ Ready-to-use, expressive visual modeling language.

❐ Extensibility and specialization mechanisms to extend the core concepts.

❐ Independent of particular programming languages and development process.

❐ Formal basis for understanding the modeling language.

❐ Encourage the growth of the OO tools (CASE) market.

❐ Support higher-level development concepts such as collaboration, frameworks, patterns, and components.

❐ Integrate best practices.

Outside the Scope of UML

❑ *Programming Languages*

- UML is a modeling language, not programming.

- Its aim is not to capture all necessary constructs of programming languages.

❑ *Tools*

- UML defines a semantic metamodel, not an tool interface, storage, or run-time model.

  – The UML specification do include some tips to tool vendors on implementation choices, but do not address everything needed.
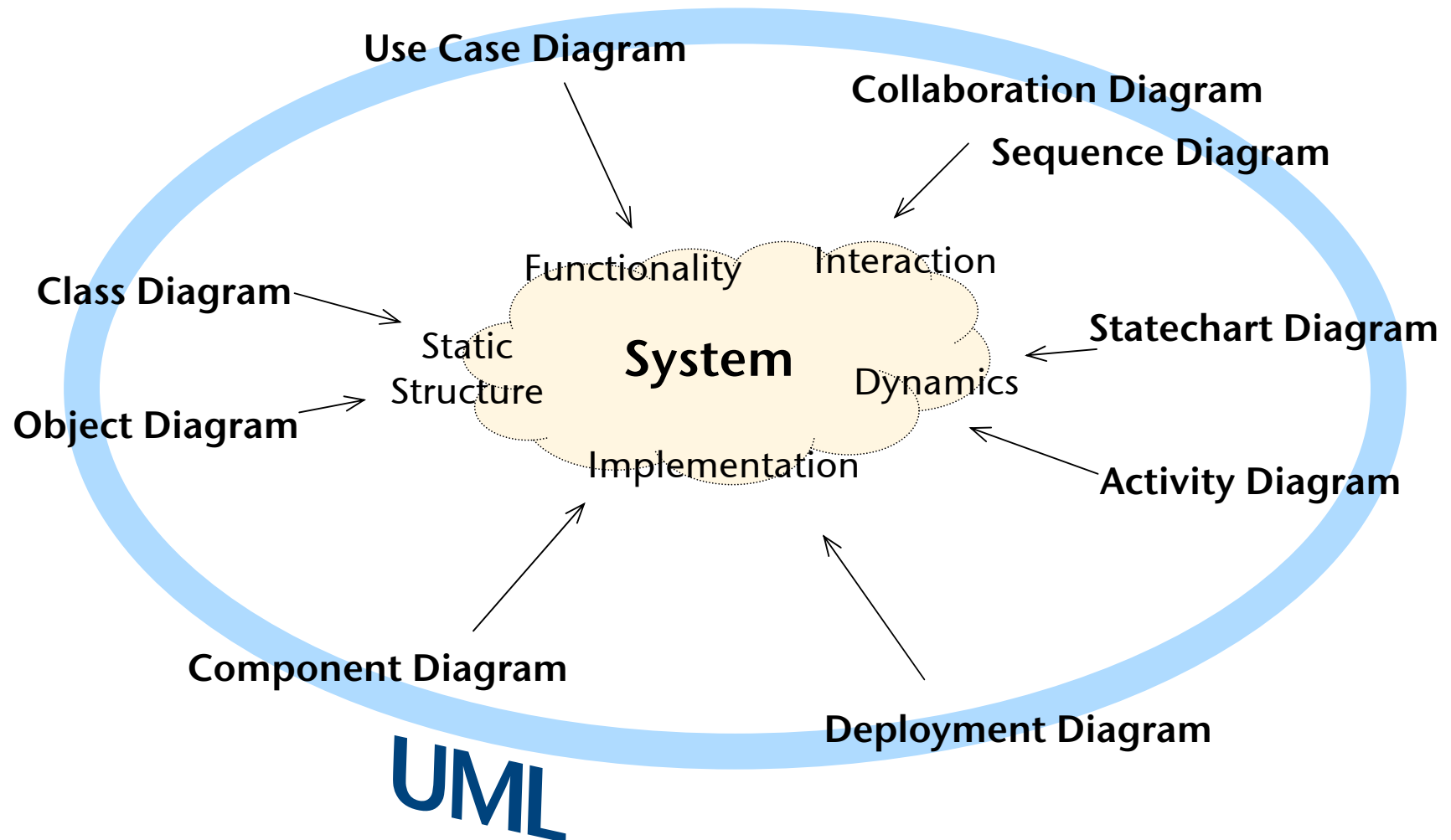
❑ *Process*

- UML is intentionally process independent, and defining a standard process was not a goal of the UML.

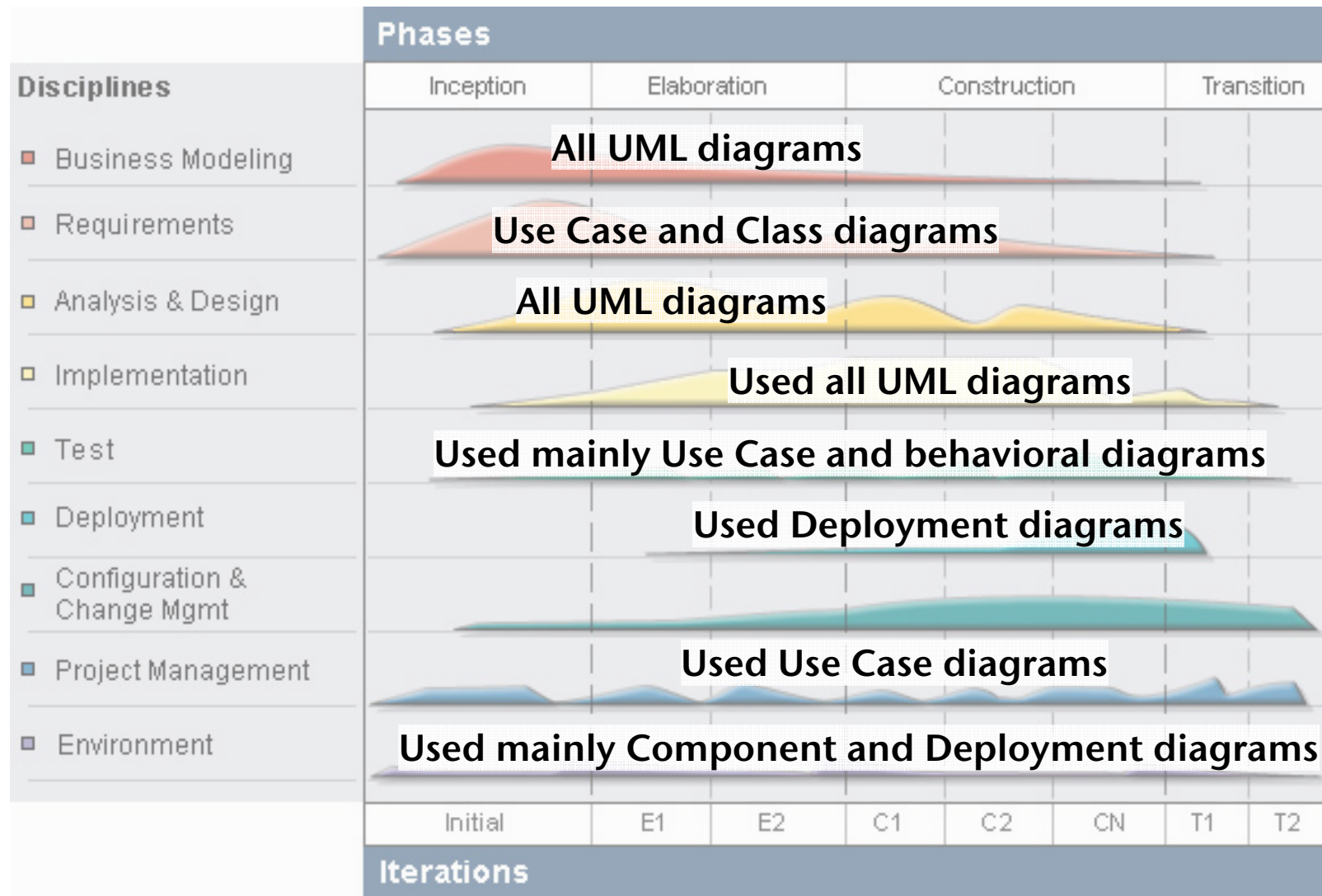- A common language for project artifacts, developed in the context of different processes.

Artifacts

❑ *Unified Modeling Language: Infrastructure*

- Defines the abstract foundational language constructs required for UML specification.

- UML meta-metamodel (MOF layer M3).

❑ *Unified Modeling Language: Superstructure*

- Defines the UML language, i.e., abstract syntax (metamodel), notation, and semantics.

- UML metamodel (MOF layer M2).

❑ Related OMG specifications:

- Object Constraint Language (OCL)

- XML Metadata Interchange (XMI)

- Diagram Interchange (DI)

- Human-Usable Textual Notation (HUTN)

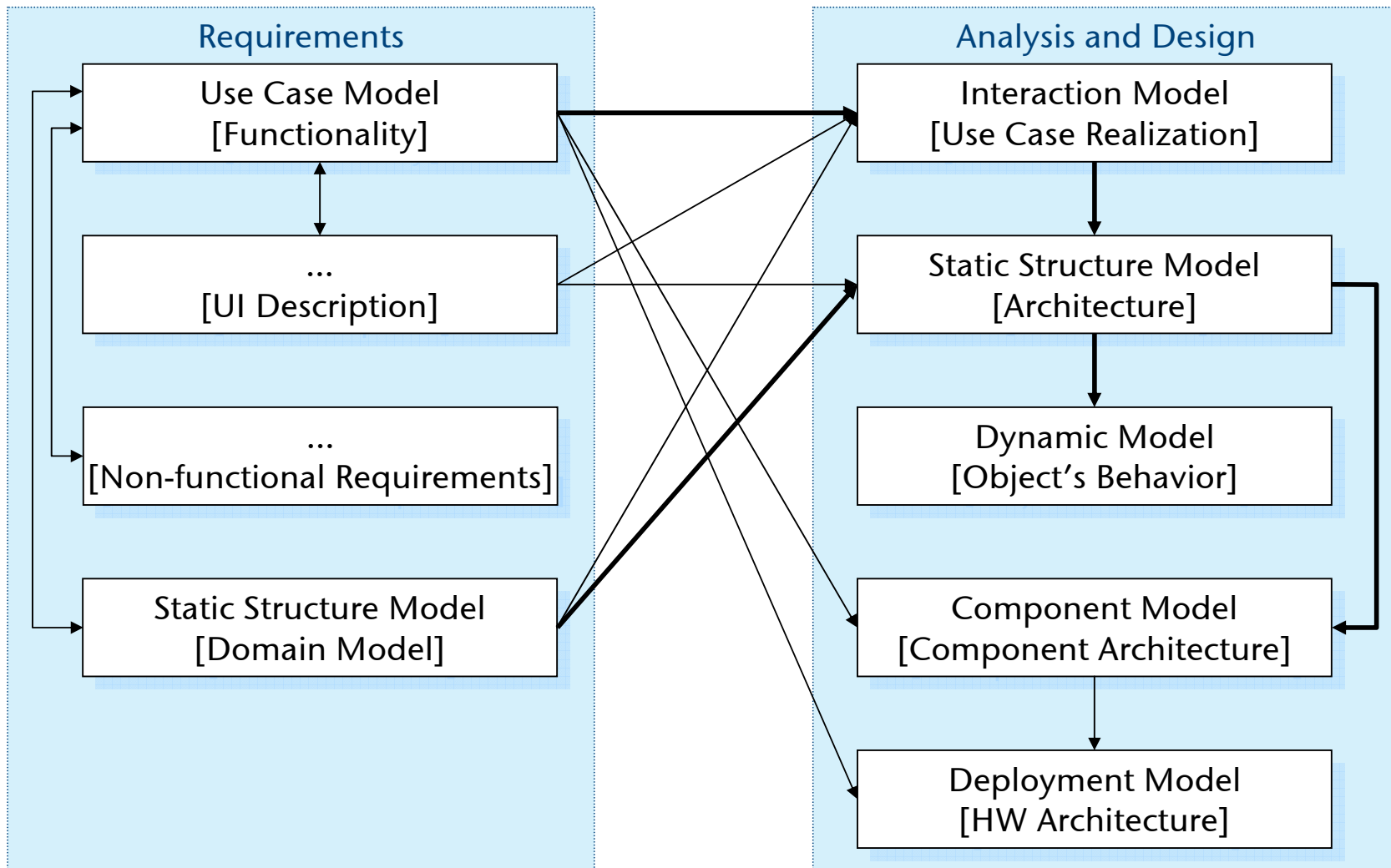- Standardized UML profiles

- Meta Object Facility (MOF)

UML and Different System's Aspects

UML in a Software Development Process

	Inception	Elaboration	Construction	Transition

Phases

Disciplines

- Business Modeling — **All UML diagrams**
- Requirements — **Use Case and Class diagrams**
- Analysis & Design — **All UML diagrams**
- Implementation — **Used all UML diagrams**
- Test — **Used mainly Use Case and behavioral diagrams**
- Deployment — **Used Deployment diagrams**
- Configuration & Change Mgmt
- Project Management — **Used Use Case diagrams**
- Environment — **Used mainly Component and Deployment diagrams**

Initial	E1	E2	C1	C2	CN	T1	T2

Iterations

Evolvement of Models

Requirements

Use Case Model [Functionality]	
... [UI Description]	
... [Non-functional Requirements]	
Static Structure Model [Domain Model]	

Analysis and Design

Interaction Model [Use Case Realization]
Static Structure Model [Architecture]
Dynamic Model [Object's Behavior]
Component Model [Component Architecture]
Deployment Model [HW Architecture]

Object-Oriented Analysis and Modeling

Generic UML Modeling Mechanisms

Model, Element, Diagram and Element View

Model

→ A set of modeling elements and diagrams used to represent the relevant aspects of the modeled system.

• Specialized package.

Element

→ A fundamental constituent of a model.

• An abstract common superclass for all metaclasses in UML.

Diagram

→ Graphical representation of parts of the UML model.

• UML diagrams contain graphical elements (nodes connected by paths) that represent elements in the UML model.

Element view

→ Graphical representation of a single element depicted in a diagram.

• One element can have several views, possibly placed in different diagrams.

Taxonomy of UML Diagrams
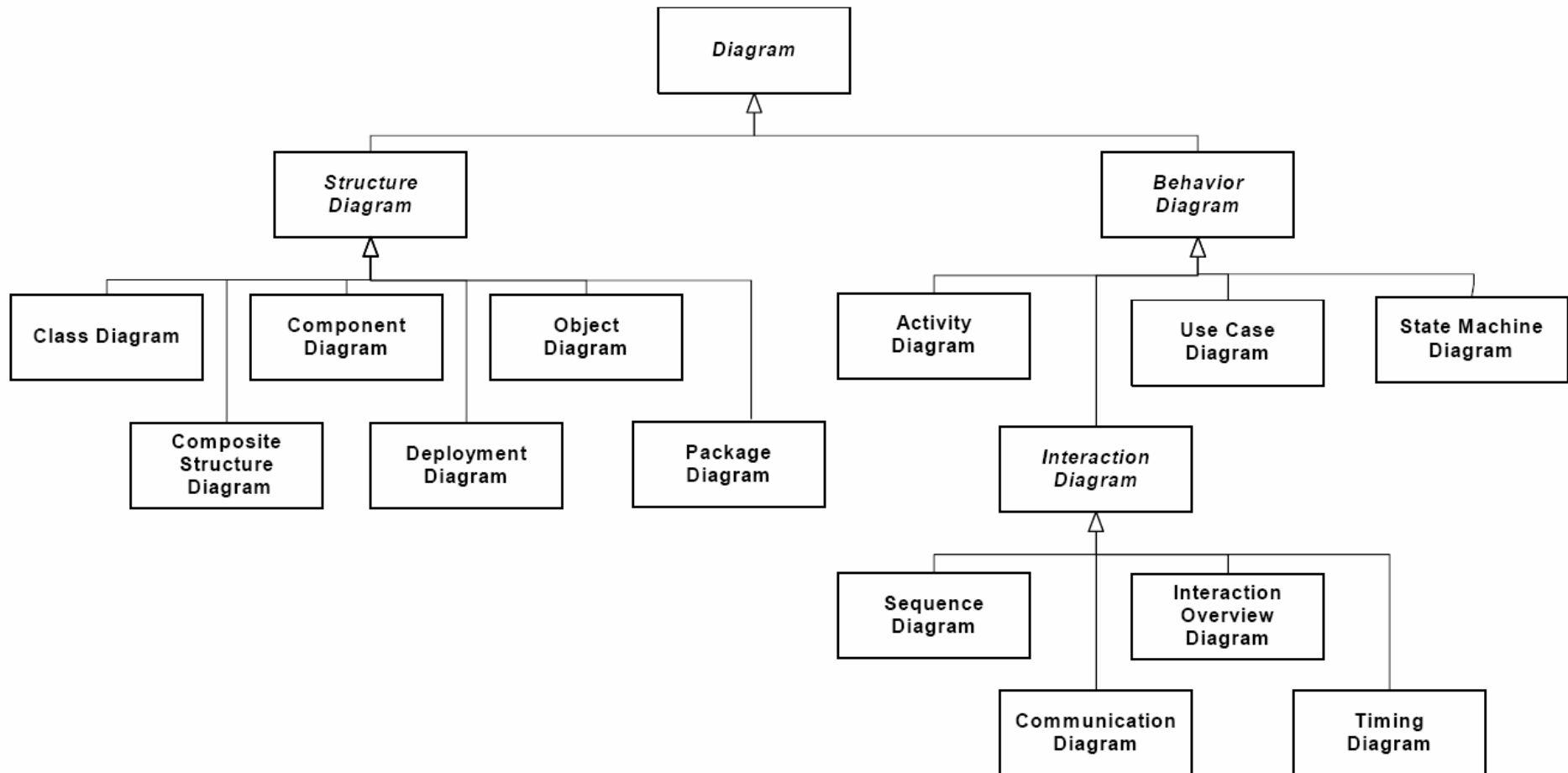
Diagram Elements

→ Generic notational mechanisms used in various ways in other parts of the language.

Graphs, Drawing Paths, Hyperlinks, …
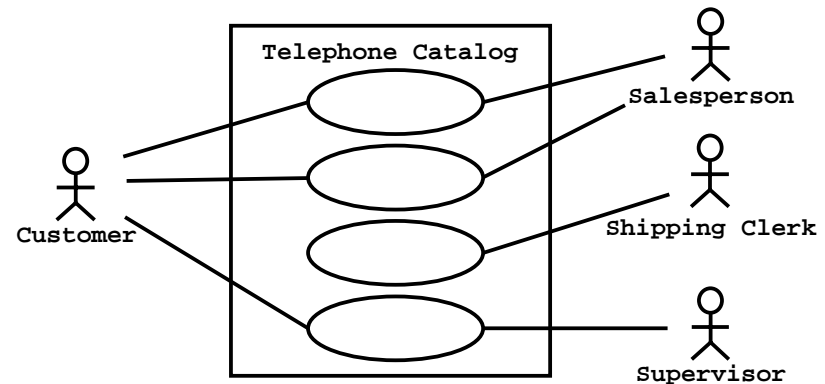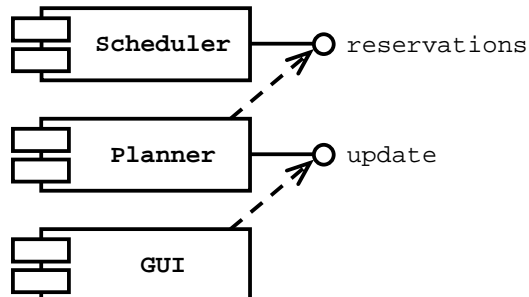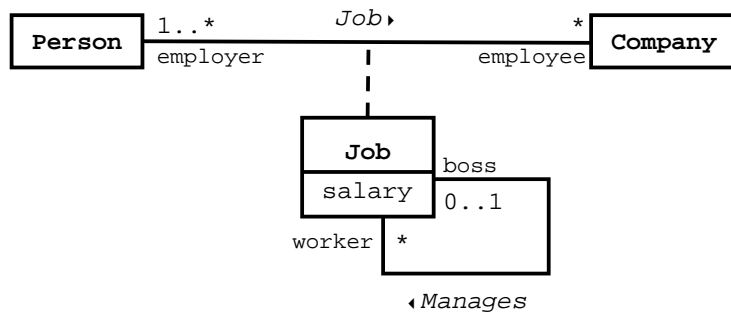
Graphs and their content

- UML diagrams are mainly graphs.

- Information is mostly in the topology.

- Graphical constructs: icons, 2-d symbols, paths and strings.

Drawing Paths

→ A series of line segments whose endpoints coincide.

Invisible hyperlinks and the hole of tools

- Arrangement of model information into a "hyperdocument".

- Dynamic notation is specific for a particular tool.

- Out of the scope of UML.

Background information

- Suppression of a model/element information.

- Textual or tabular format of background information.

- Out of the scope of UML.

String, Name and Label

String

→ A sequence of characters (of any character set).

Name

→ A string uniquely identifying a *named element*.

• Defined within a *namespace*.

• May be linked together by delimiters into a pathname.

```
BankAccount, controller, long_underscored_name,
MathPack::Matrices::BandedMatrix.dimension
```

Label

→ A string that is attached to a graphical symbol.

```
BankAccount
```

account ← label

Keyword and Expression
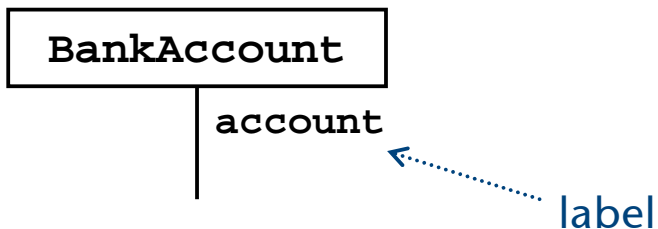
Keyword

→ A name reserved by UML.

● Used in stereotypes and tagged values.

«keyword»

{keyword}

Expression

→ Linguistic formulas that yield values when evaluated in run-time.

● Language-dependent.

```
BankAcount
BankAccount * (*) (Person*, int)
array [1..20] of range(-1.0 .. 1.0) of Real
[i > j and self.size > i]
```

Copyright © 2007 by Whitestein Technologies, Switzerland

Comment

→ A textual annotation that can be attached to a set of elements.

This model was created by
Rowan Atkinson.

Bla, bla,
bla,...

Customer
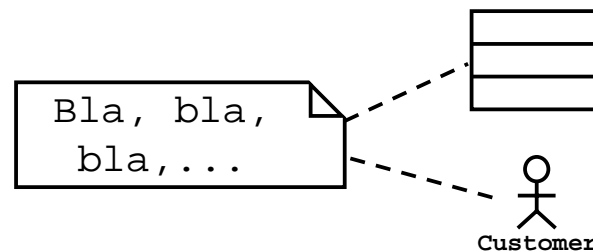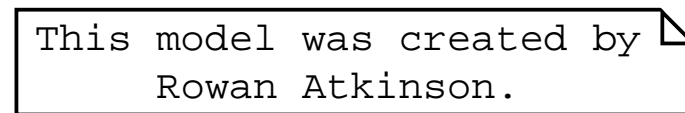
Constraint

→ A condition or restriction expressed in natural language text or in a machine readable language (e.g. OCL) for the purpose of declaring some of the semantics of an element.

{name: boolean-expression}
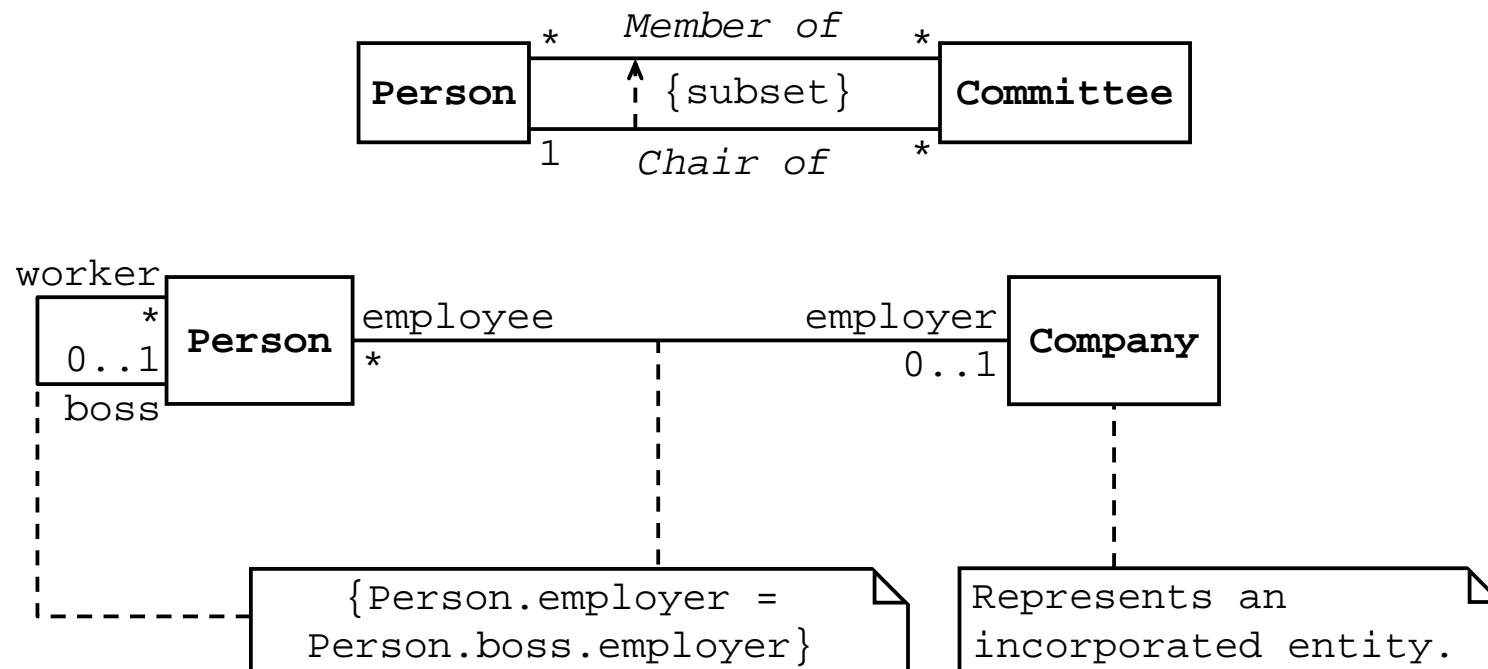
Tagged Values (Property String)

→ A set of keyword-value pairs attached to a model element.

❑ Keyword (tag) identifies the type of a property.

❑ Value determines the property's value.

❑ If the type is Boolean and the value is omitted $\Rightarrow$ True

❑ Can be used as an element in a list.

- It applies to all subsequent elements.

{keyword = value, keyword = value, ... }

```
{author="John",deadline=15-June-98,status=Design}
{abstract}
```

Stereotypes

→ A new type of modeling element introduced at modeling time.

→ Specialization (a special meaning) of existing modeling element type with the same form but a different intent/semantics.

❏ Can be used with any standard UML element type.

❏ Enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended element types.
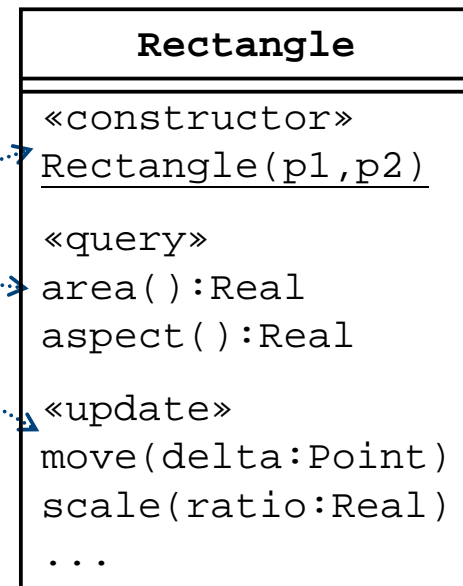
❏ Can be used as an element in a list.

- It applies to all subsequent elements.

«stereotype name»

and/or an icon

Examples of Stereotypes

```
┌─────────────────────┐    ┌─────────────────────┐    ┌─────────────────────┐
│     «control»       │    │     «control»   ↻   │    │              ↻      │
│     PenTracker      │    │     PenTracker      │    │      PenTracker     │
└─────────────────────┘    └─────────────────────┘    └─────────────────────┘
```

            ↻

        PenTracker   protocols

```
                                    ┌──────────────────────────┐
                                    │         Rectangle        │
                                    ├──────────────────────────┤
                                    │ «constructor»            │
                                    │ Rectangle(p1,p2)         │
                                    │                          │
                                    │ «query»                  │
                                    │ area():Real              │
                                    │ aspect():Real            │
                                    │                          │
                                    │ «update»                 │
                                    │ move(delta:Point)        │
                                    │ scale(ratio:Real)        │
                                    │ ...                      │
                                    └──────────────────────────┘
```

```
┌─────────────┐   «calls»    ┌─────────────┐
│ JobManager  │- - - - - - ->│  Scheduler  │
└─────────────┘              └─────────────┘
```

Classifier-Instance Correspondence
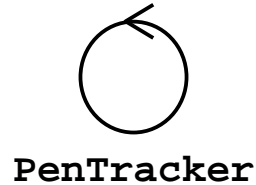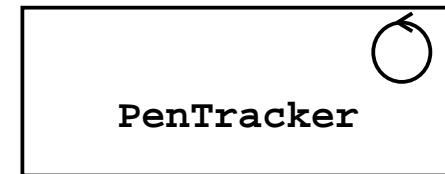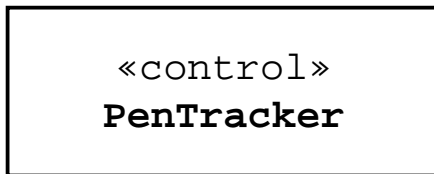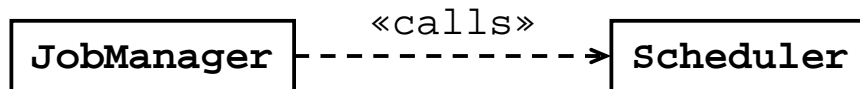
Classifier

→ A classification of instances, it describes a set of instances that have features in common.

Instance

→ An instance in a modeled system. It can be classified by one or more classifiers.

❐ Dual form of modeling elements: classifier and instance.

❐ Notation of the instance form uses the same geometrical symbol as the classifier but name is underlined .

❐ Examples: class-instance specification, association-link, parameter-value, operation-call, ...

Dependency

→ A relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation.

❏ The complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).

«dependencyType»

| Client | - - - - - - - - - - - - - → | Supplier |

| Client | | Supplier |

| Client | | Supplier |
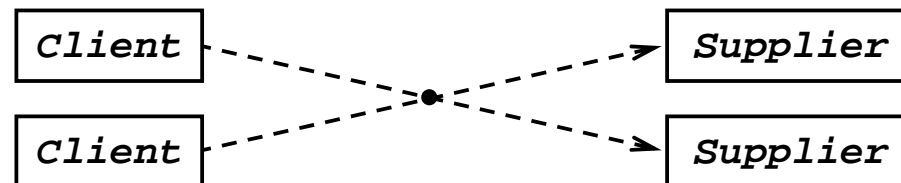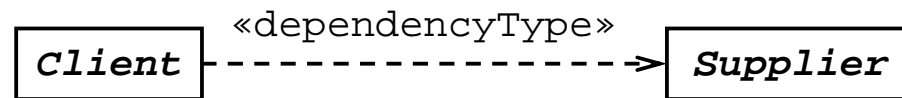
Examples of Dependencies
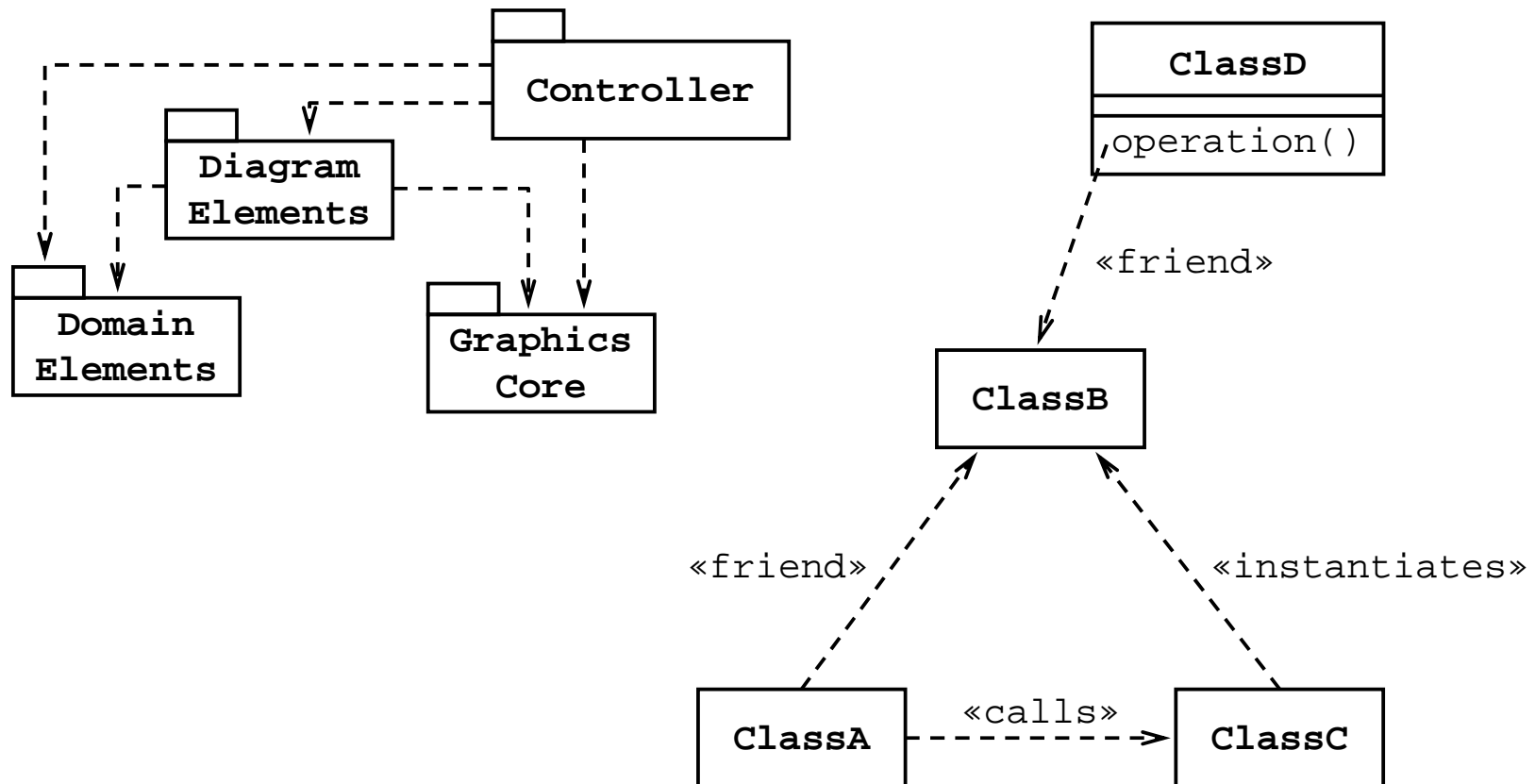
Package

→ A package is used to group elements, and provides a namespace for the grouped elements.

❏ Only packageable elements can be owned members of a package.

❏ May contain other packages.

❏ Package can be used in:

- Use Case View ⇒ functional decomposition
- Static Structure View ⇒ logical high-level architecture
- Component View ⇒ modular decomposition
- Deployment View ⇒ physical hardware decomposition

❏ Structure diagrams containing only packages are called *Package Diagrams*.

Standard Types of Packages
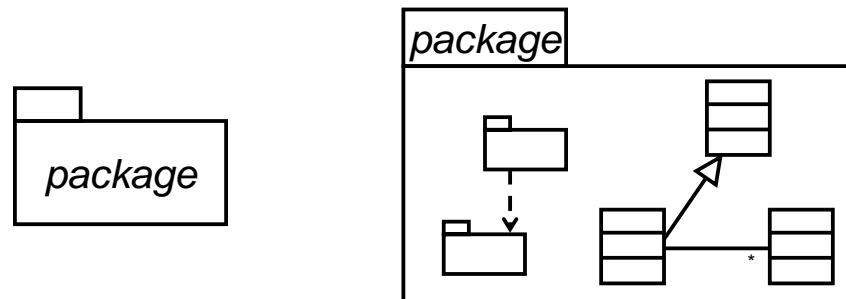
❑ «framework»

→ A package that contains model elements that specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns, or templates. When frameworks are specialized for an application domain they are sometimes referred to as application frameworks.

❑ «modelLibrary»

→ A package that contains model elements that are intended to be reused by other packages. A model library is analogous to a class library in some programming languages.

Example of Package Diagram
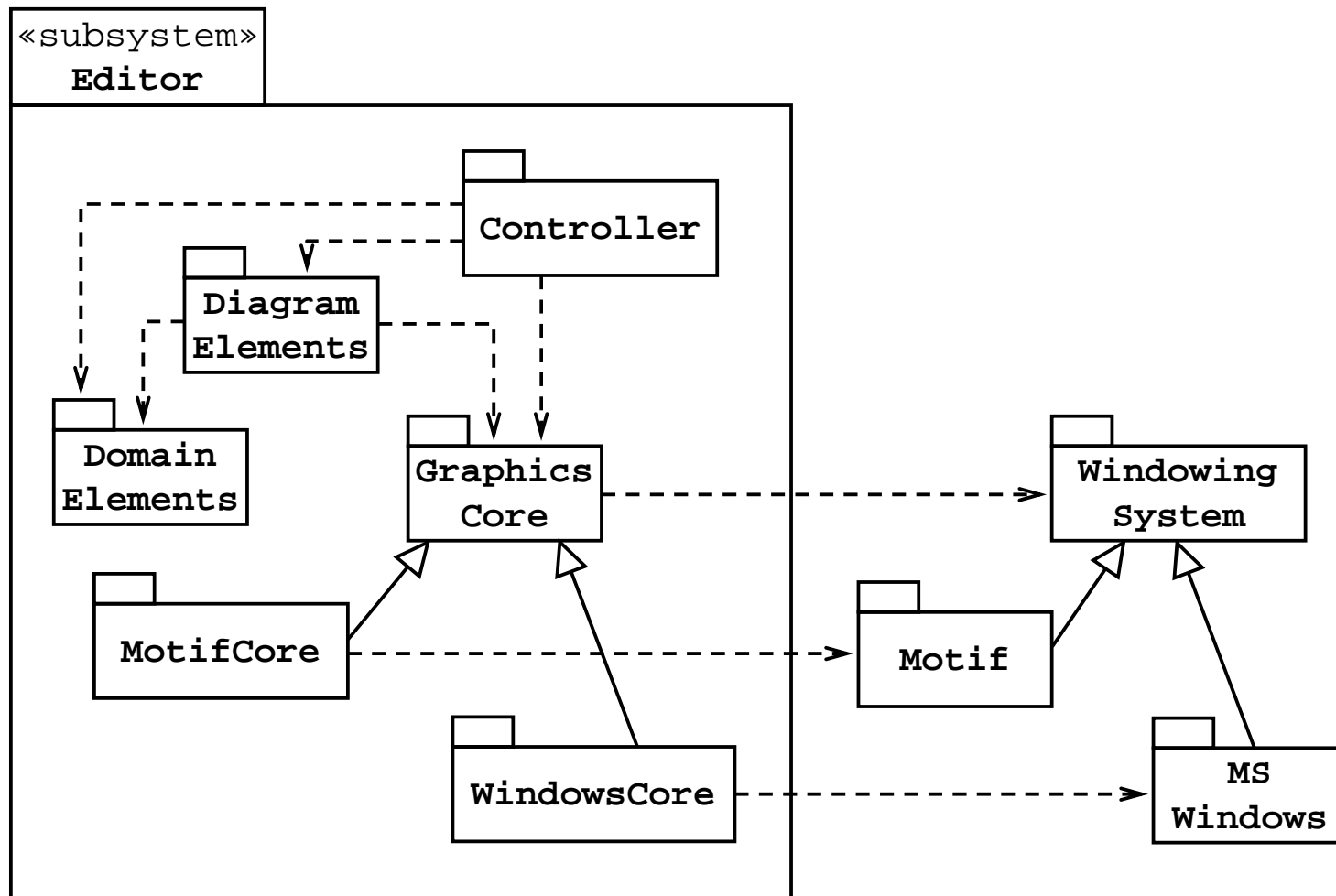
Example Package Composition

(Packageable) Element and its Visibility

Packageable Element

- → A named element that may be owned directly by a package.
- Must always have a visibility.

Visibility Kind

- Public '+'

  - → A public element is visible to all elements that can access the contents of the namespace that owns it.

- Private '-'

  - → A private element is only visible inside the namespace that owns it.

- Protected '#'

  - → A protected element is visible to elements that have a generalization relationship to the namespace that owns it.

- Package '~'

  - → A package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace.

Package Import

→ A directed relationship that allows the use of unqualified names to refer to package members from other namespaces.

❒ The import visibility–visibility of the imported packageable elements within the importing namespace.

- Can only be public (the default value) or private.

- If the package import is public, the imported elements will be visible outside the package, while if it is private they will not.

❒ Notation: binary dependency relationship with stereotype «import».

❒ Full element identification: *package*:: ... ::*package*::*element*

```
┌─────────────────────────────┐        ┌───────────────────────────┐
│ Customers    │              │        │ Banking   │               │
├──────────────┘              │        ├───────────┘               │
│ ┌─────────────────────────┐ │«import»│ ┌───────────────────────┐ │
│ │ Banking::CheckingAccount│ │- - - - ▶│ │ CheckingAccount       │ │
│ └─────────────────────────┘ │        │ └───────────────────────┘ │
│                             │        │                           │
└─────────────────────────────┘        └───────────────────────────┘
```

importing package   imported package
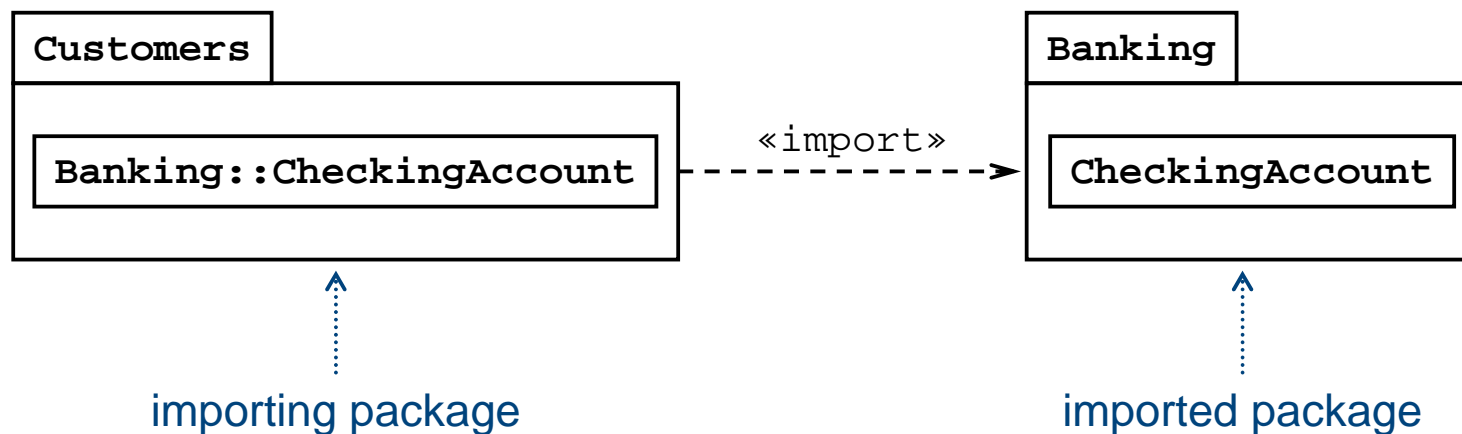
Package Merge

→ A directed relationship between two packages that indicates that the contents of the two packages are to be combined.

❑ "Package generalization".

❑ Extending/specialization of elements with same names in source and target packages.

❑ Transformation rules and constraints for the contained packages, classes, data types, properties, associations, operations, constraints, enumerations, and enumeration literals.

```
┌──────┐                        ┌──────┐
│  Source  │ - - - «merge» - - →│  Target  │
└──────────┘                    └──────────┘
```
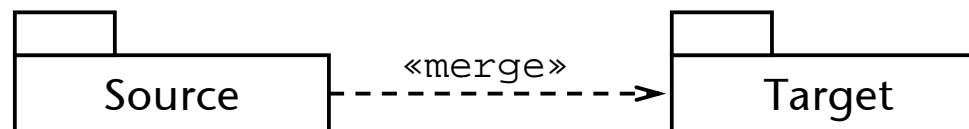
Conceptual View of the Package Merge
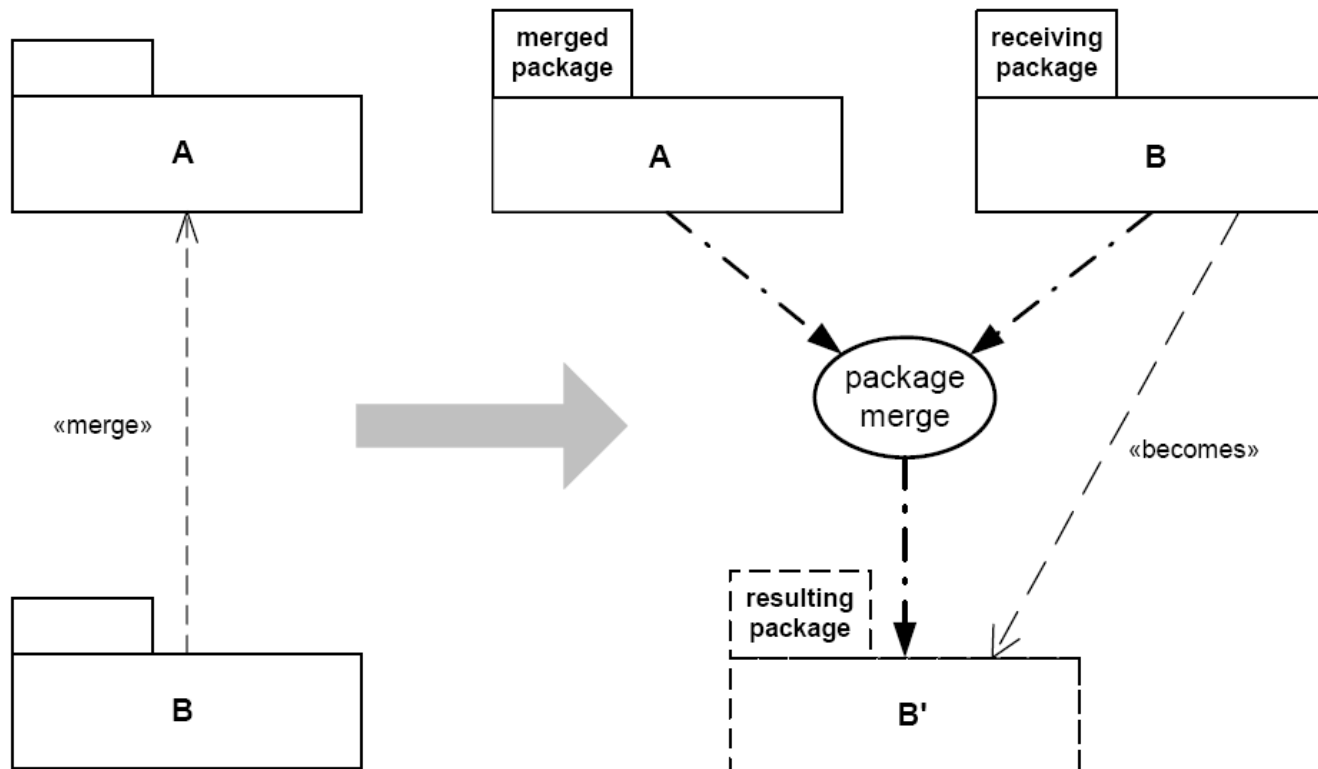
Examples of Package Merge

merge of P1::A into P2::A
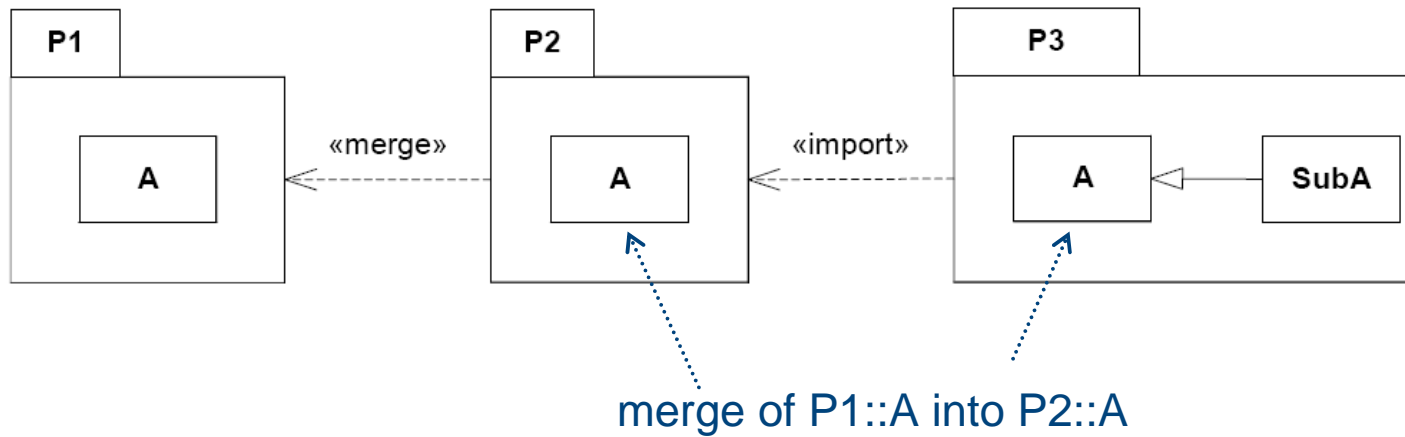
Examples of Package Merge (cont.)
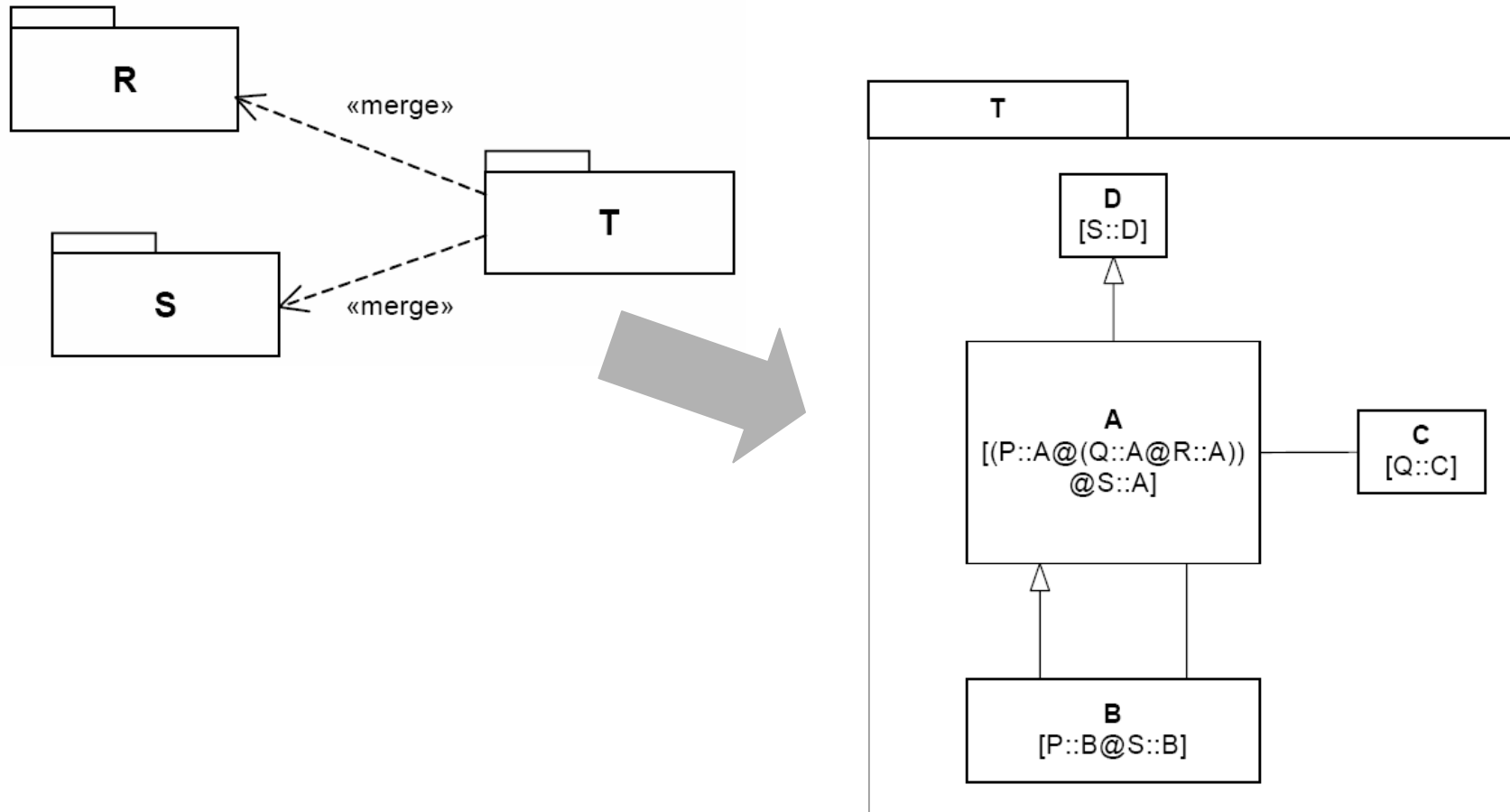
Object-Oriented Analysis and Modeling

Use Case Modeling

Use Case Model

→ **Functionality of the system and its surroundings.**

Consists of:

- Use Case diagrams
- Use Case and Actor descriptions

Supported by:

- UI descriptions (incl. prototypes)
- Specification of non-functional requirements

Used (mainly) in:

- Requirements $\Rightarrow$ functional system requirements
- Analysis and design$\Rightarrow$ analysis and design models
- Testing $\Rightarrow$ test cases
- Management $\Rightarrow$ planning and tracking

Use Case Diagram

❏ Defines outer behavior/functionality/required usages of a system.

❏ *Actors*, *use cases*, *subjects* and their relationships.
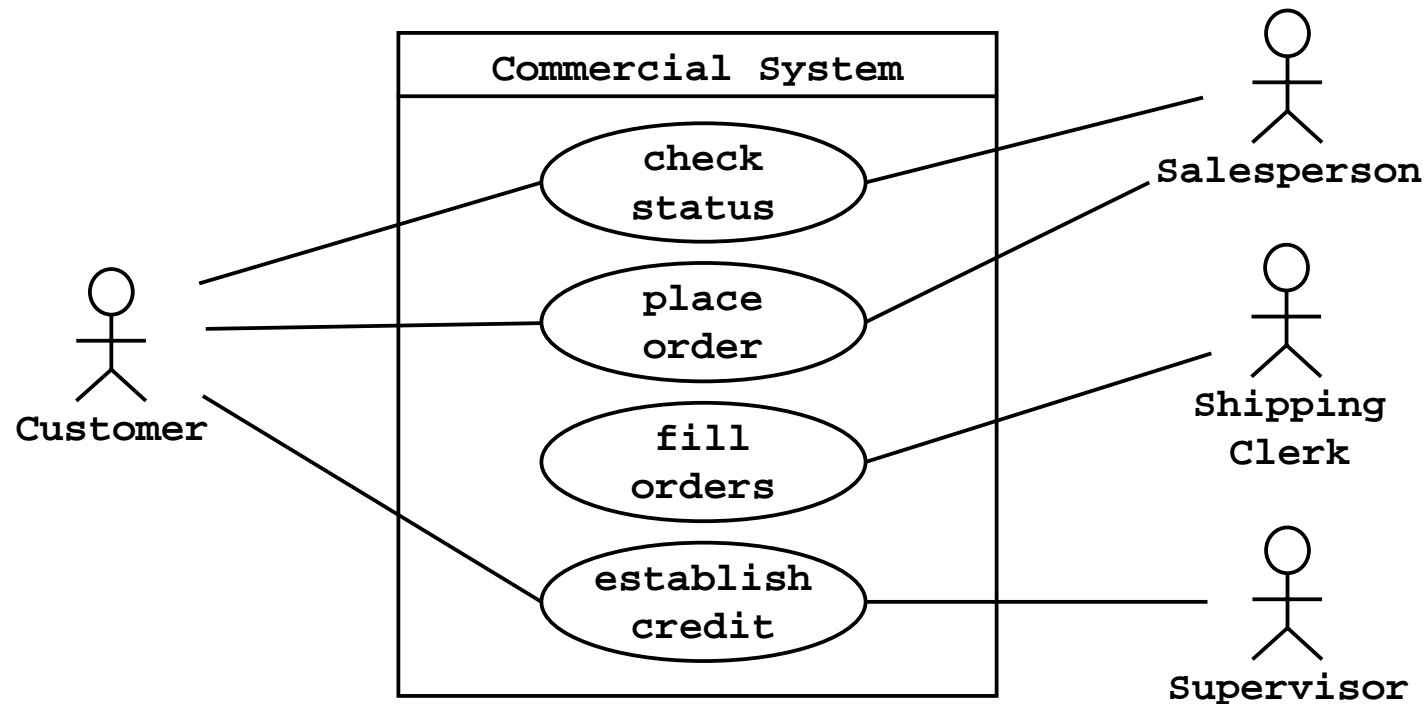
Actor

→ A role played by an entity that interacts with the subject (system).

❒ External to the subject.

❒ Represent roles played by human users, external hardware, or other subjects.

❒ A single physical instance may play the role of several different actors

❒ A given actor may be played by multiple different instances.

❒ A specialized classifier (behaviored classifier).

Customer

```
        «actor»
       Customer
```

Directory Server

Use Case
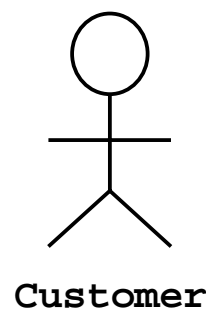
→ The specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.

❑ A coherent behavior, possibly including variants (exception of error handling), that the subject can perform in collaboration with one or more actors.

- Primary actor initiates the use case.

- Secondary actor is used to complete the use case.

❑ Can be internally described by interactions, activities, and state machines, or by pre-conditions and post-conditions as well as by natural language text where appropriate.

❑ A specialized classifier (behaviored classifier).

Withdraw

Fill orders

extension points
Selection

Perform ATM Transaction

OrderStationery

extension points
order created: in Created state
order shipped: in Shipped state
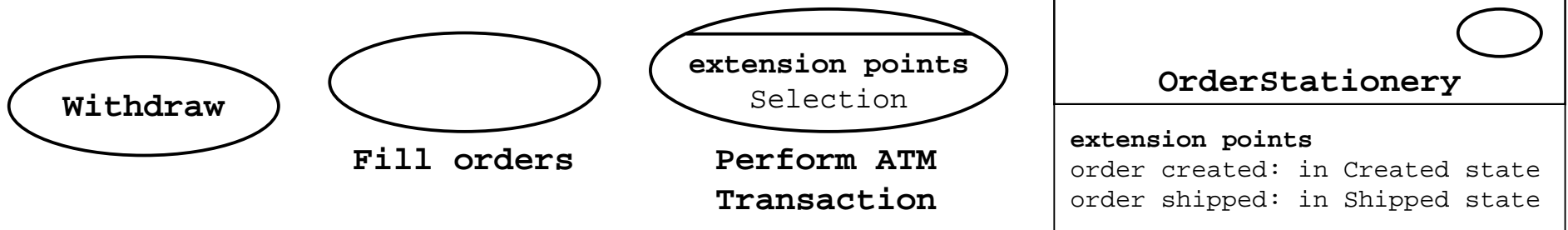
Scenario

→ A session that an actor instance has with the system.

❑ Has details of real data and actual expected output.

❑ Potentially hundreds to thousands in an application.

1
> **John enters his account# 404504**
> **John enters his pin# 9342**
> **John requests his average balance from 1/1/97 - 7/31/97**
> **System gives the average balance**

2
> **Larry enters his account# 4343443**
> **Larry enters his pin# 84954**
> **Larry requests his average balance from 1/1/95 - 12/31/96**
> **System gives the average balance**

3
> **Mary enters her account# 34334**
> **Mary enters her pin# 4343**
> **System detects non valid pin and repeats the procedure**

Use Case vs. Scenario

Use cases are not scenarios!!!

❑ Use case represents a set of potential scenarios.

❑ Looking at a family of similar scenarios, you can gather the essence of what is typically done.

❑ Similar scenarios will follow similar patterns of work and provide similar types of results.

❑ Normally each use case focuses on a specific goal.

- E.g. to obtain the current account balance.

Scenario → **Use Case**

Use Case Description
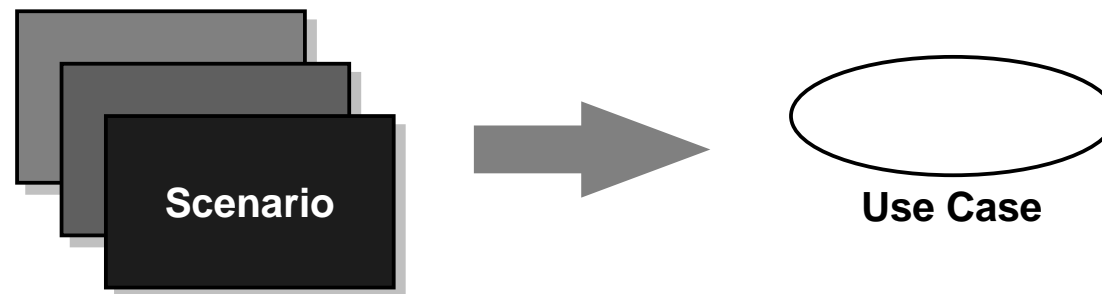
CHARACTERISTIC

- ❑ **Name:** use case name

- ❑ **Goal:** a longer statement of the goal

- ❑ **Scope:** subsystem, application, ...

- ❑ **Pre-conditions:** state before use case execution

- ❑ **Post-conditions:** state after use case execution

- ❑ **Trigger:** action upon which is use case started

- ❑ **Primary actor:** a role name

- ❑ **Secondary actors:** list of other needed roles or systems

ALGHORITHM (primary scenario, extensions and variations)

- ❑ steps of the algorithm:
  - • <step#> <action description>

- ❑ control expressions:
  - • if then else, repeat, switch, ...

RELATED INFORMATION

- ❑ **Priority:** how critical

- ❑ **Time:** a performance duration

- ❑ **Frequency:** how often

- ❑ **Supplementary specifications:** other non-functional requirements

System (UML 1.*)

→ The functional boundary of the system.

❑ Described by a finite set of use cases.

❑ Actors are drawn outside of the system, use cases inside.
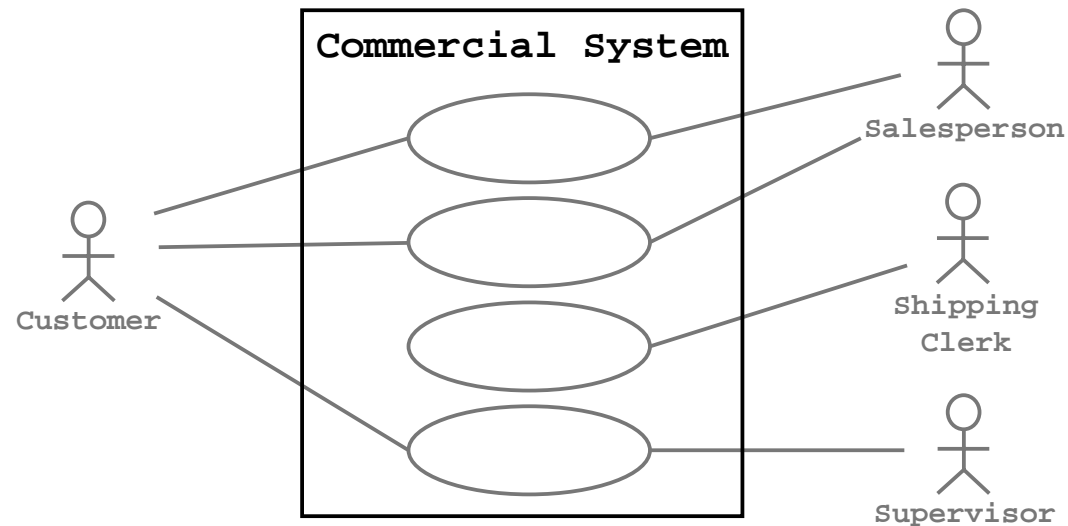
Classifier as a Subject (UML 2.*)

❑ A classifier with the capability to own use cases.

❑ Typically represents the subject to which the owned use cases apply.

```
┌─────────────────────────────┐
│       DepartmentStore       │
├─────────────────────────────┤
│                             │
│        ╭───────────╮        │
│       (  MakePurchase )     │
│        ╰───────────╯        │
│                             │
└─────────────────────────────┘
```
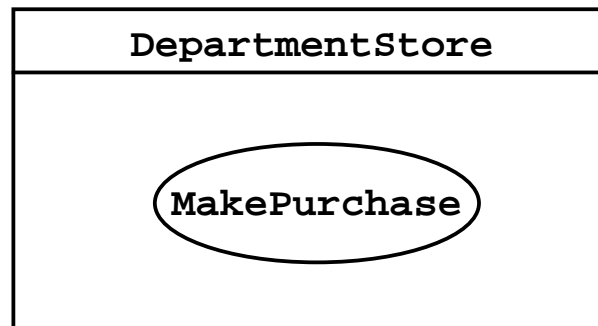
Association (in Use Case Diagrams)

→ A (binary) relationship between an actor and a use case meaning interaction of the actor with the use case's subject.

❑ Multiplicity

  → The range of allowable instances at the association end.

  ● A list of values and intervals expressed as: lower limit .. upper limit

  ● Possible values : number or * (many)

    '1', '*', '0..*', '1..*', '2, 4, 6..10, 20..*'

❑ Navigability

  → Indicates that navigation is supported toward attached classifier.
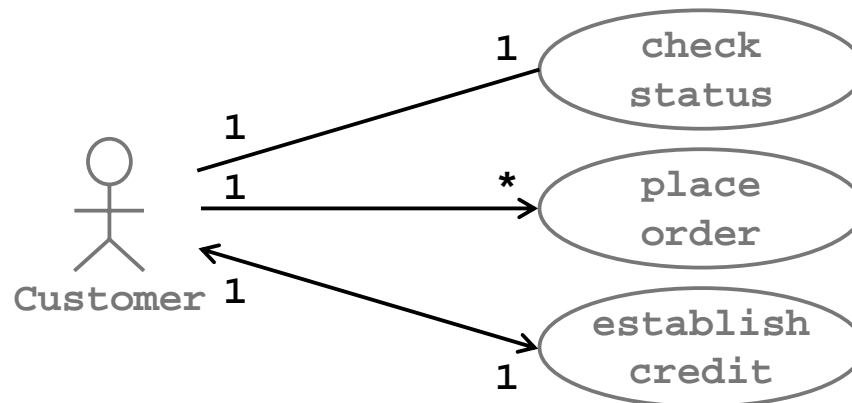
Include

→ Indicates that an *including* use case contains the behavior defined in another, *included* (base), use case.

❏ Included use case obviously implements a behavior shared by a number of use cases ("subroutine").

❏ The included use case is always required for the including use case to execute correctly.
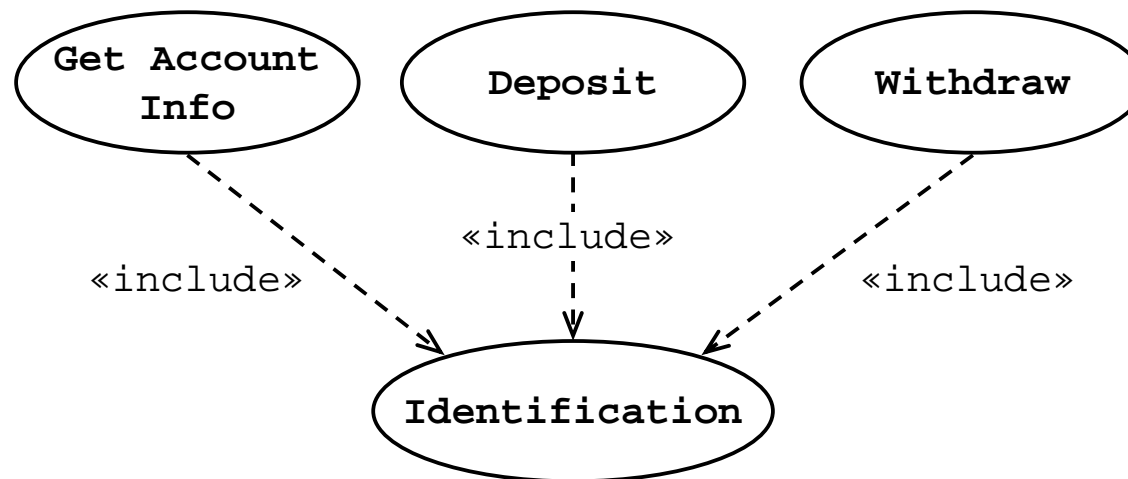
Extend

→ A relationship from an *extending* use case to an *extended* use case that specifies how and when the behavior defined in the extending use case <u>can</u> be inserted into the behavior defined in the extended use case.

❒ The extension takes place at one or more specific extension points defined in the extended use case.

❒ The extended use case is defined independently of the extending use case.

❒ The extending use case usually depends on the extended use case.

❒ Extension often represents unusual behavior, such as an exception or error handling.

```
Condition: {customer selected HELP}
extension point: Selection
```

«extend»

extension points
Selection

**Perform ATM
Transaction**

Online Help
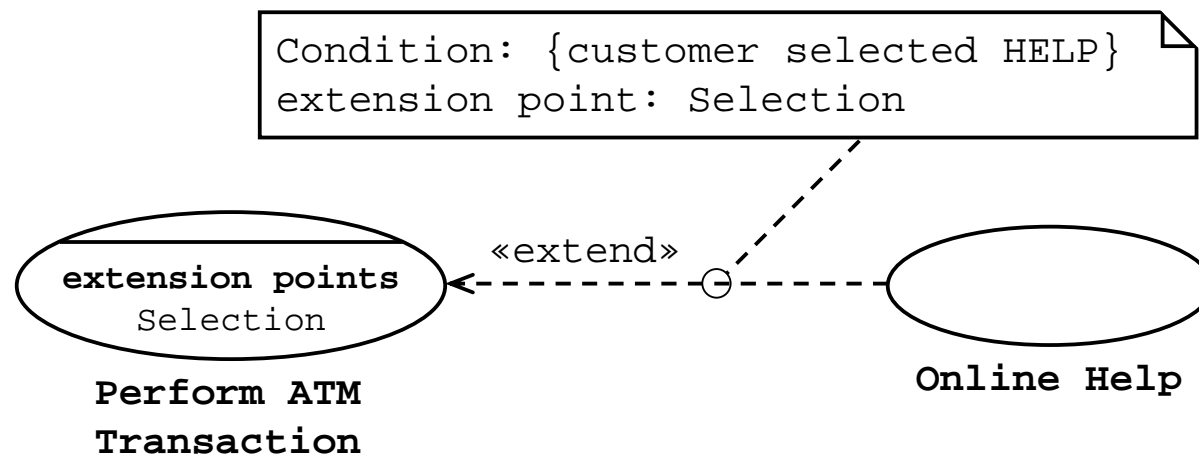
Example of Extension

Create New Employee - extended use case

1. Enter employee information
2. Enter salary information
3. Enter job title
4. Save entry
5. **Validation:** System validates
6. If no problems, systems setups new employee record

Create Body Guard - extending use case

- System checks employee police record

extension point: Validation

extension points
Validation

Create New Employee

«extend»

Create Body Guard
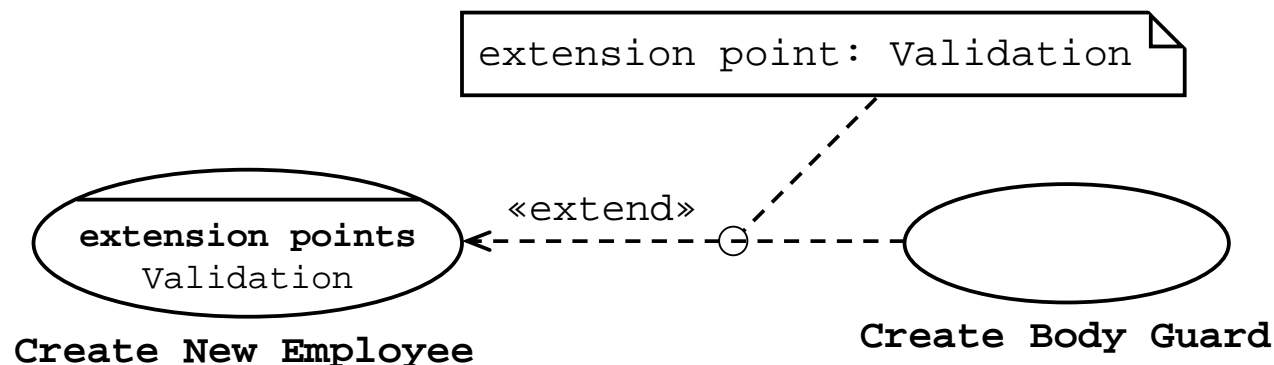
Generalization of Actors

- The specific (child) actor gets all capabilities of the general (parent) actor.

- Simplifies the diagram, without losing semantics.

- Abstract actors can also be used.
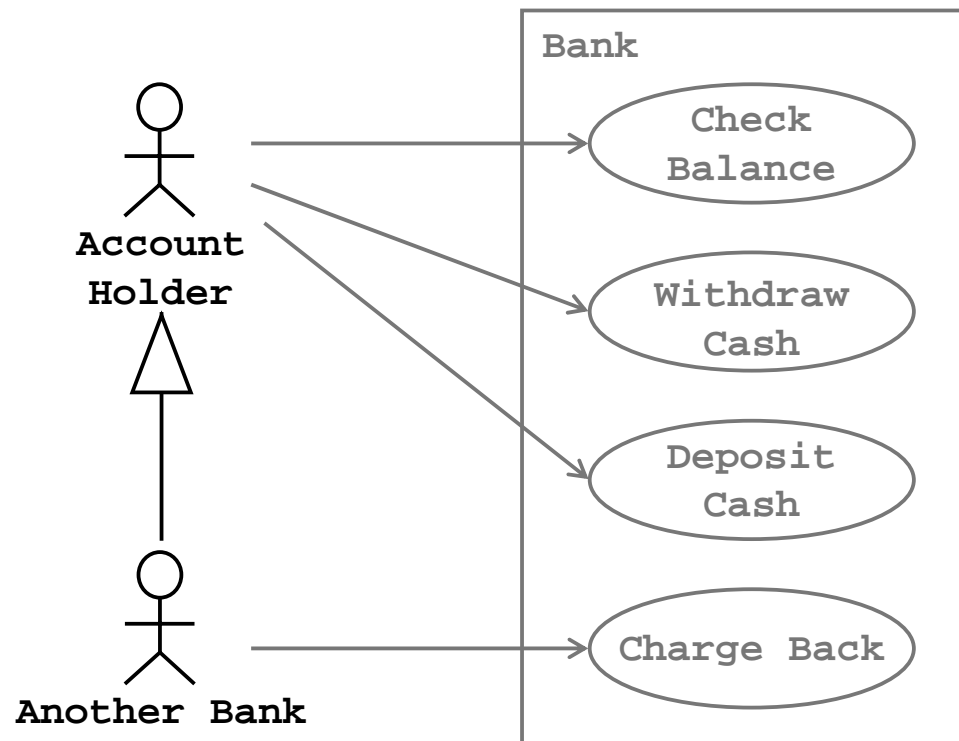
Generalization of Use Cases

□ The specific use case represents a more specific behavior than the behavior of the general use case.

□ Abstract use cases can also be used.
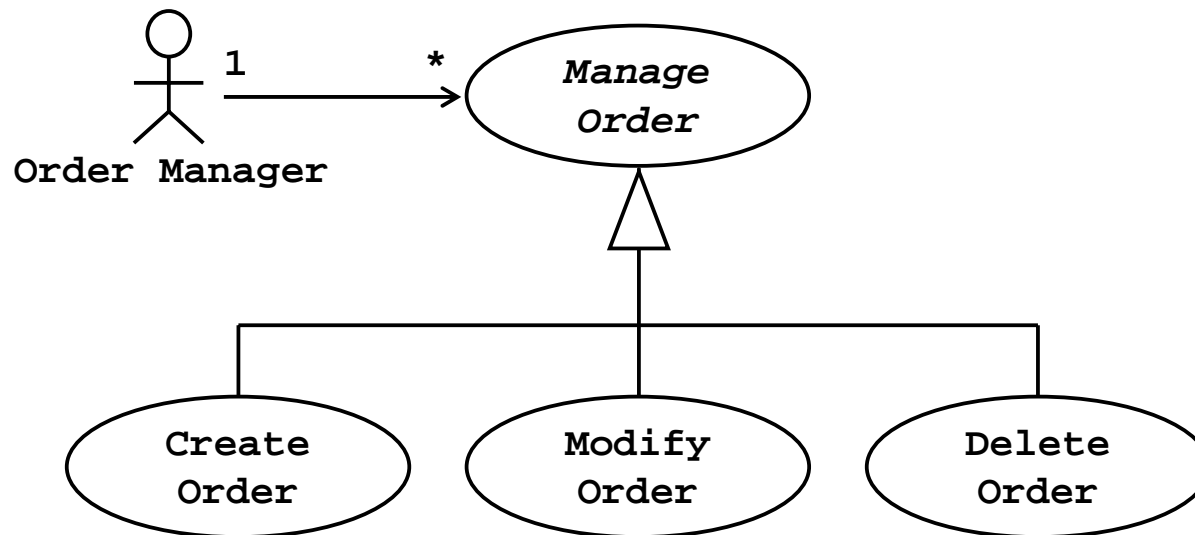
Process of Use Case Modeling

1. Capture a common vocabulary (Glossary).

2. Find actors.

3. For each actor determine a set of "its" use cases.

4. Briefly describe use cases and actors.

5. Package use cases and actors.

6. Present the use case model in use case diagrams.

7. Structure the use case model.

8. Prioritize use cases.

9. Describe use cases.

10. Review the use case model.

Rules for Use Case Diagrams

❑ Differentiate primary and secondary actors.

❑ Use case must provide a real service to user.

❑ Keep drawings clear and neat.

 • Do not put too many use cases in one diagram.

 • Simpler diagrams are easier to understand.

❑ Split up the use case model into use case packages.

 • Package related use cases.

 • *Functional decomposition* of the system.

Object-Oriented Analysis and Modeling

Classes

Class (Structural) Model

→ **Structure of the system expressed in terms of classes, interfaces, objects and their relationships.**

Consists of:

- Class diagrams.

- Object diagrams.

- Package diagrams.

- Element descriptions.

Supported by:

- State machines.

- Activities.

- Interactions.

Used (mainly) in:

- Requirements $\Rightarrow$ domain/conceptual model.

- Analysis $\Rightarrow$ analytical (logical) model.

- Design $\Rightarrow$ design model.

Diagrams

- *Structure Diagram*

  - → An abstract diagram type showing the static structure of the objects in a system.

  - • Has several specific diagrams: class diagram, object diagram, composite structure diagram, component diagram, deployment diagram, and package diagram.

- *Class Diagram*

  - → Classes, interfaces and their relationships.

- *Object Diagram*

  - → Static structure of instances (objects and links).

  - • A snapshot of the state of the system at a point in time.

  - • Possibly compatible with a particular class diagram.

- *Package Diagram*

  - → Packages and their relationships.

Perspectives

- ❑ *Conceptual*

  - Conceptual/domain model.

  - No (little) regard for the SW implementation.

  - Used in Requirements.

- ❑ *Specification*

  - Logical application model.

  - Focused on software.

  - Concerning on types rather than implementation.

  - Used in Analysis.

- ❑ *Implementation*

  - Implementation model.

  - Used in Design.

Example of Class Diagram
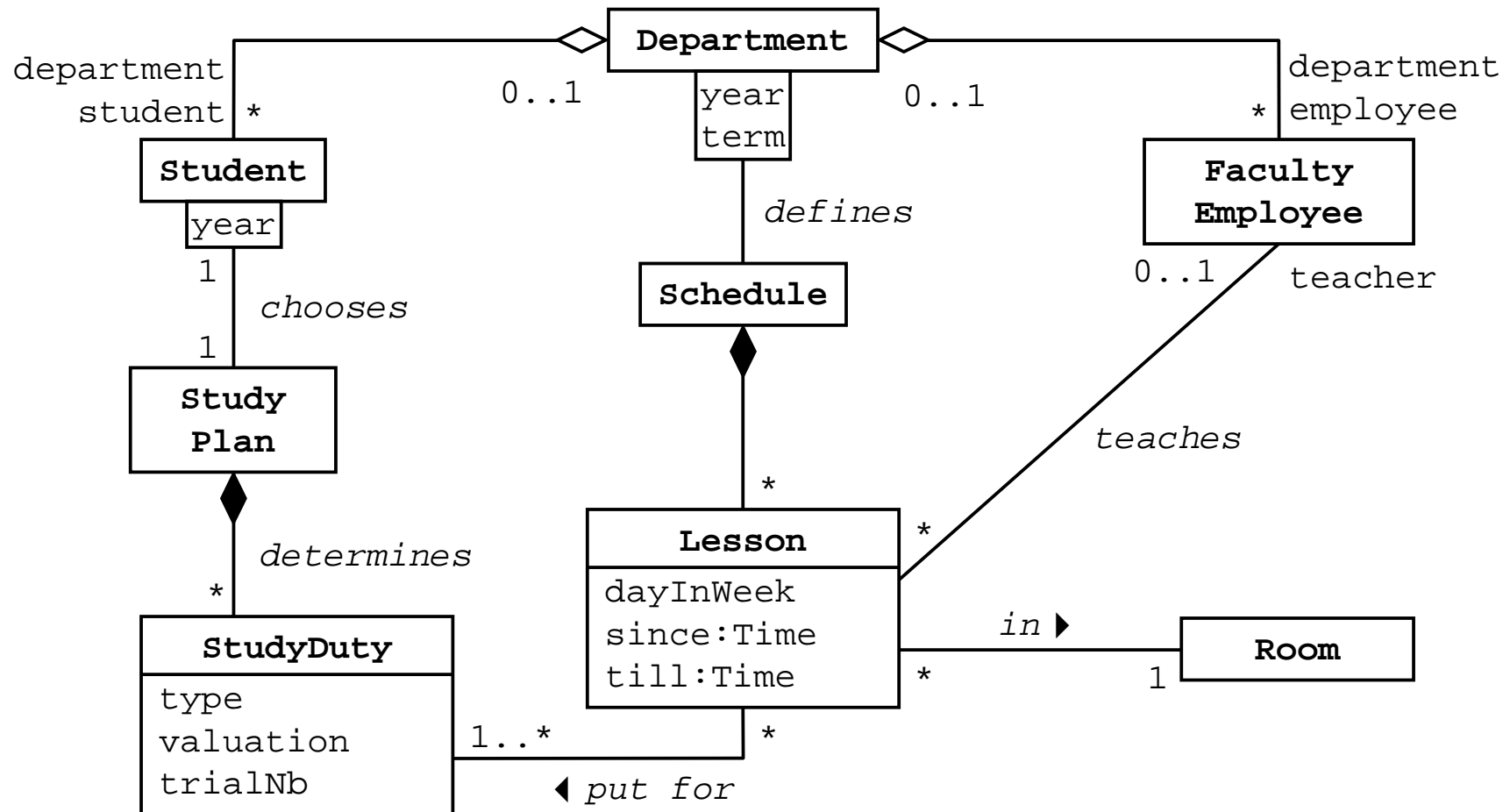
Class

→ An abstraction of set of objects that share a common structure (attributes, operations and links) and a common behavior/semantics.

❑ A kind of classifier whose features are attributes and operations.

❑ Format of attributes (owned properties):

property ::= [*visibility*] ['/'] *name* [':' *type*] ['[' *multiplicity* ']'] ['='*default*]
   ['{' *prop-modifier* [',' *prop-modifier*]* '}']

- Visibility: '+' public, '-' private, '#' protected, '~' package

- Derived property, which can be computed from other properties, is marked by '/'.

- Multiplicity:

  – positive number (0, 1, 2, ...)

  – interval: *lower-bound* '..' *upper-bound*

  – '*' for infinite upper bound

  – examples: 3, 1..4, 1..*, *

name
attribute list
operation list

Class (cont.)

- Property modifier:

  - 'readOnly' means that the property is read only.

  - 'union' means that the property is a derived union of its subsets.

  - 'subsets' *property-name* means that the property is a proper subset of the property identified by *property-name*.

  - 'redefines' *property-name* means that the property redefines an inherited property identified by *property-name*.

  - 'ordered' means that the property is ordered.

  - 'unique' means that there are no duplicates in a multi-valued property.

  - *prop-constraint* is an expression that specifies a constraint that applies to the property.

Class (cont.)

❐ Format of operations:

[*visibility*] *name* '(' [*parameter-list*] ')' [':' [*return-type*]
  ['{' *oper-property* [',' *oper-property*]* '}']]

- Parameters:

  parameter-list ::= *parameter* [',' *parameter*]*

  parameter ::= [*direction*] *parameter-name* ':' *type-expression* ['['*multiplicity*']']
    ['=' *default*]
    ['{' *parm-property* [',' *parm-property*]* '}']

  – direction: 'in', 'out', 'inout' (defaults to 'in' if omitted)

- Operation properties (modifiers):

  – 'redefines' *oper-name* means that the operation redefines an inherited operation identified by *oper-name*.

  – 'query' means that the operation does not change the state of the system.

  – 'ordered' means that the values of the return parameter are ordered.

  – 'unique' means that the values returned by parameters have no duplicates.

  – *oper-constraint* is a constraint that applies to the operation.

❐ Class (static) attributes and operations are underlined.

Examples of Classes

```
┌──────────┐
│  Window  │
└──────────┘
```

```
┌─────────────┐
│  UI::Window │
└─────────────┘
```

```
┌──────────────────────┐
│       Window         │
├──────────────────────┤
│ size: Area           │
│ visibility:Bool      │
├──────────────────────┤
│ display()            │
│ hide()               │
└──────────────────────┘
```

```
┌──────────────────────────────────┐
│             Window               │
├──────────────────────────────────┤
│ +size:Area=(100,100)             │
│ #visibility:Bool=invisible       │
│ +defaultSize:Rectangle           │
│ -xptr:*Xwindow                   │
├──────────────────────────────────┤
│ +Window():Window                 │
│ +display(dpy:Display=':0.0')     │
│ +hide()                          │
└──────────────────────────────────┘
```
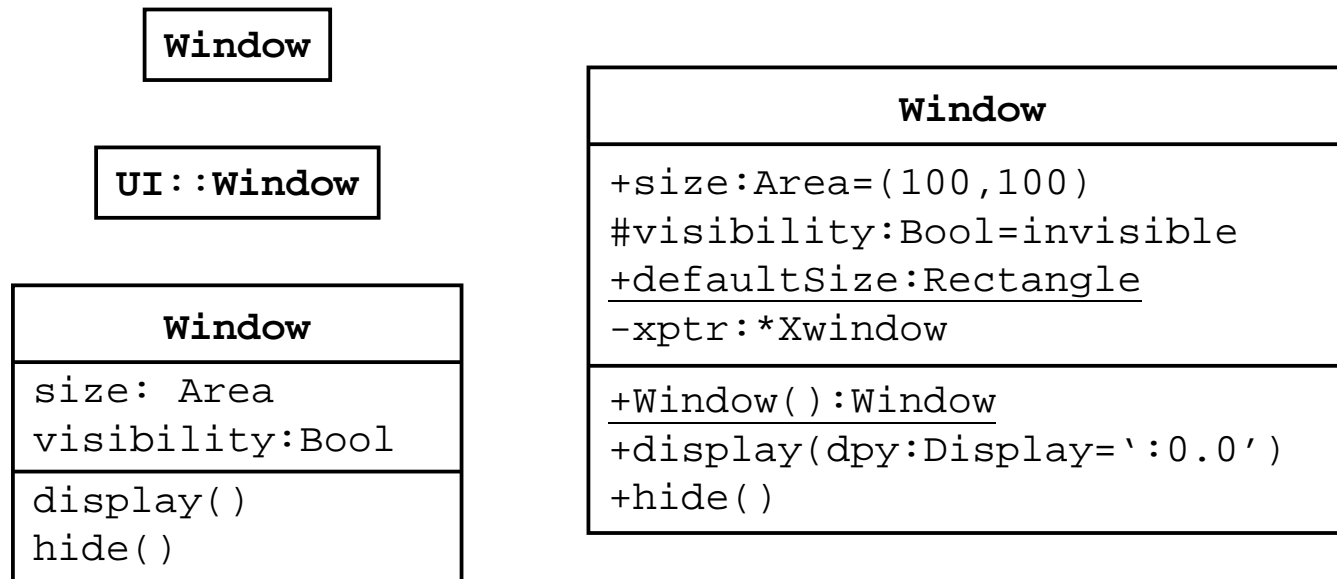
Example of Derived Attribute

Person
birthday
/age

{age = currentDate - birthday}
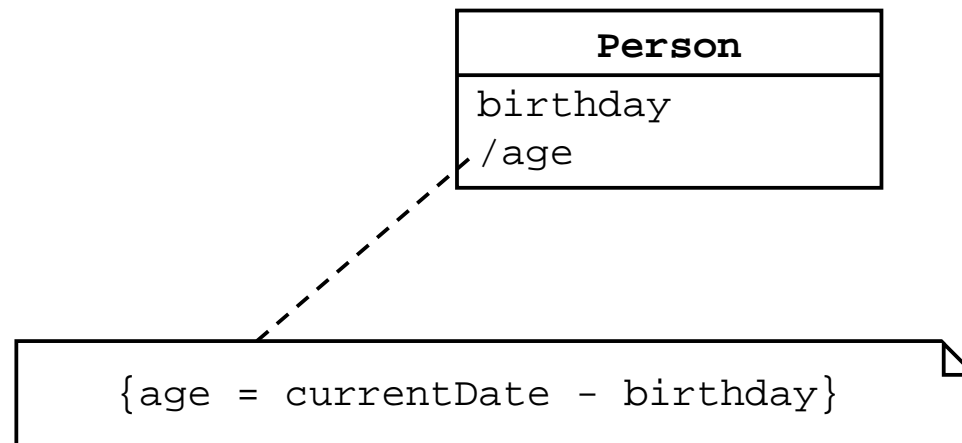
Interface

→ A kind of classifier that represents a declaration of a set of coherent public features and obligations.

❐ Specifies a *contract*; any instance of a classifier that realizes the interface must fulfill that contract.

❐ An interface is ***not instantiable***; instead, an interface is *implemented* by an instantiable classifier, which means that the instantiable classifier presents a public facade that conforms to the interface specification.
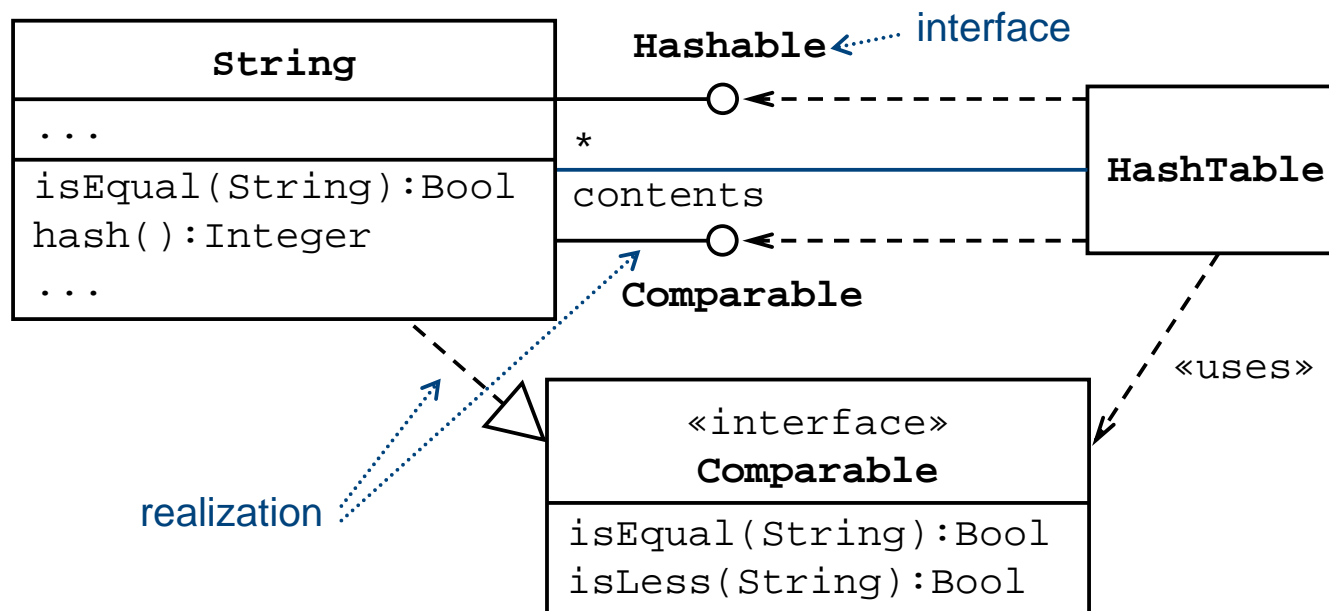
Association

→ A relationship that can occur between typed instances.

❑ An association declares that there can be *links* between instances of the associated types.

- A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end.

❑ It has at least two ends represented by properties, each of which is connected to the type of the end.

❑ More than one end of the association may have the same type.

❑ Association end:

- Association role name.

- Multiplicity.

- Ownership of the end by the association: indicated by a small circle.

- Navigability:

    – navigable
    – non-navigable
    – unspecified

Association (cont.)

- ❑ Association end (cont.):

  - Visibility: +, -, #, ˜

  - Aggregation kind (only for binary associations):

    - None.

    - Shared (for aggregation):

      - A weak relationship between the *whole* and its *parts*.

      - Parts can exist independently on the whole.

      - Also called "ownership by a reference".

    - Composite (for composition):

      - A strong relationship between the *whole* and its *parts*.

      - A part instance must be included in at most one composite (whole) at a time. If a composite is deleted, all of its parts are normally deleted with it.

      - Compositions may be linked in a directed acyclic graph with transitive deletion characteristics.

Association (cont.)

- ❐ Association end (cont.):

  - Property string (enclosed in curly braces):

    - {subsets *property-name*} to show that the end is a subset of the property called *property-name*.

    - {redefines *end-name*} to show that the end redefines the one named *end-name*.

    - {union} to show that the end is derived by being the union of its subsets.

    - {ordered} to show that the end represents an ordered set.

    - {bag} to show that the end represents a collection that permits the same element to appear more than once.

    - {sequence} or {seq} to show that the end represents a sequence (an ordered bag).

  - Qualifier: an attribute or a list of attributes whose values serve to partition the set of links.

- ❐ Association and its ends may be derived; marked by '/' before their names.

Association Class

→ An association with class-like properties (attributes, operations, relations, behavior).

❏ It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not to any of the classifiers.

❏ An association and its connected association class represent the same model element.

- Therefore, they must have the same name.

Examples of Associations

association name

A ── * ── *AssociationAB* ── * ── B

endA   endB

end owned by B

association end

derived union

A ── a | 0..1 ──────── /b{union} | 0..* ──> B

{subsets b}

A ── a | 0..1 ──────── b1 | 0..* ──> B

subset end
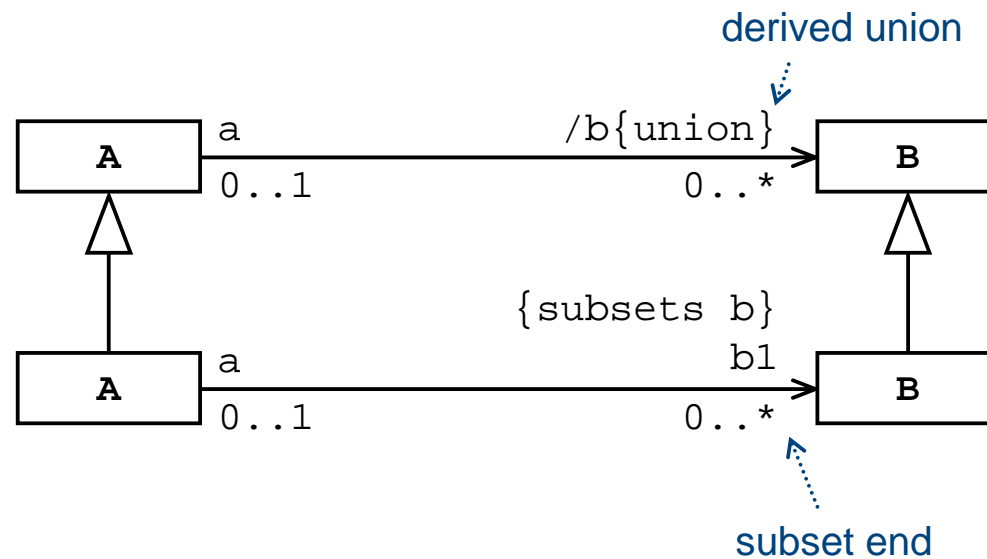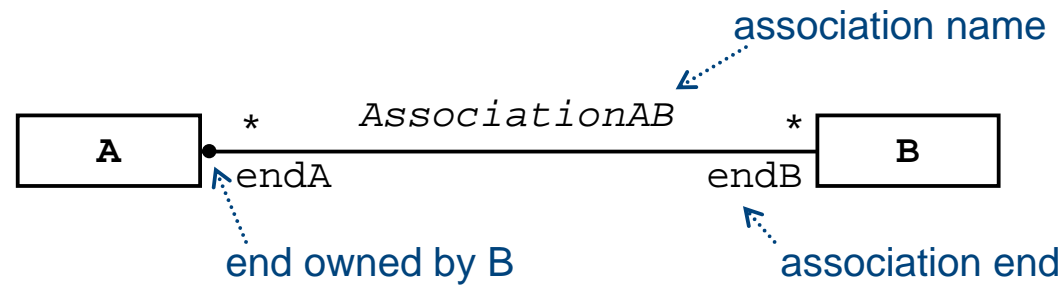
Examples of Associations and Assoc. Classes

multiplicity

Person 1..* employee — *Job* ▶ — * employer **Company**

association class → **Job** / salary / worker

boss 0..1

rolename

◀ *Manages*

reading direction

n-ary association

Year

* season

Bank / account → qualifier

*

1 → navigation

Person

Player goalkeeper * — ◇ — team 1 **Team**

Record / goalsFor / goalsAgainst / wins / losses
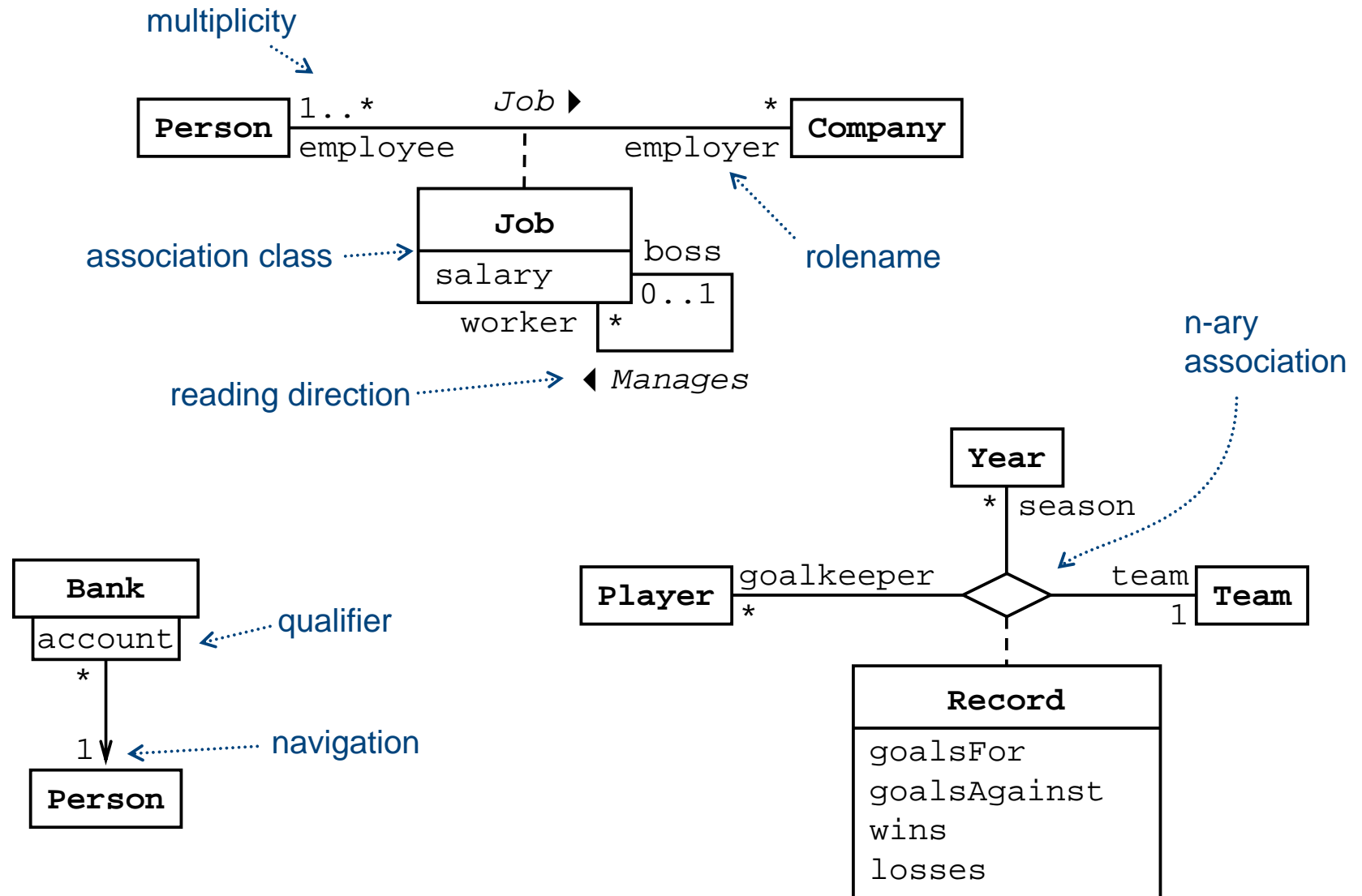
Examples of Aggregations and Compositions

Aggregation:

```
                          +points
            1  Contains ▸  3..*
  Polygon ◇——————————————————————▸ Point
            1            {ordered}

              1                        GC
Committee ◇——————— Person              ┌──────────┐
            1..5                    1  │ GC       │
            member                     ├──────────┤
  Polygon ◆                    -gc     │ color    │
            1 ┘——————————————————————▸ │ texture  │
                                       └──────────┘
```

Composition:

```
┌────────────────────────┐              ┌──────────┐
│        Window          │              │  Window  │
├────────────────────────┤              └──────────┘
│ scrollbar[2]:Slider    │          1 ◆  ◆   ◆  1
│ title:Header           │               │    1
│ body:Panel             │
└────────────────────────┘     scrollbar 2  title 1   body 1
                               ┌────────┐ ┌────────┐ ┌────────┐
                               │ Slider │ │ Header │ │ Panel  │
                               └────────┘ └────────┘ └────────┘
```
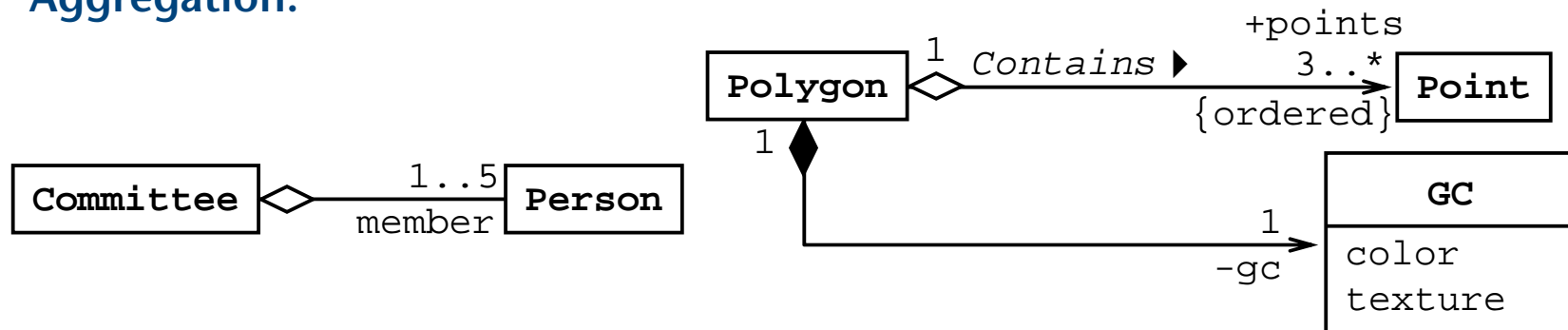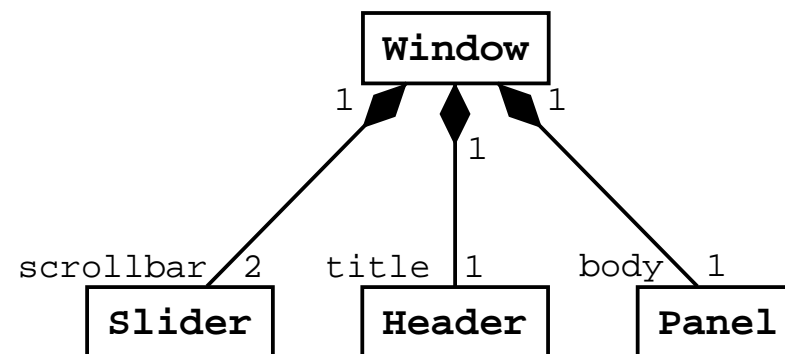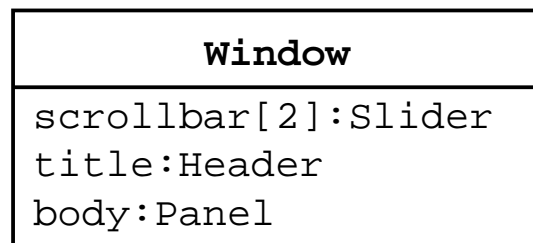
Generalization

→ The taxonomic relationship between a more general classifier and a more specific classifier.

❑ The specific classifier inherits the features of the more general classifier.

❑ Each instance of the specific classifier is also an indirect instance of the general classifier.

Generalization Set
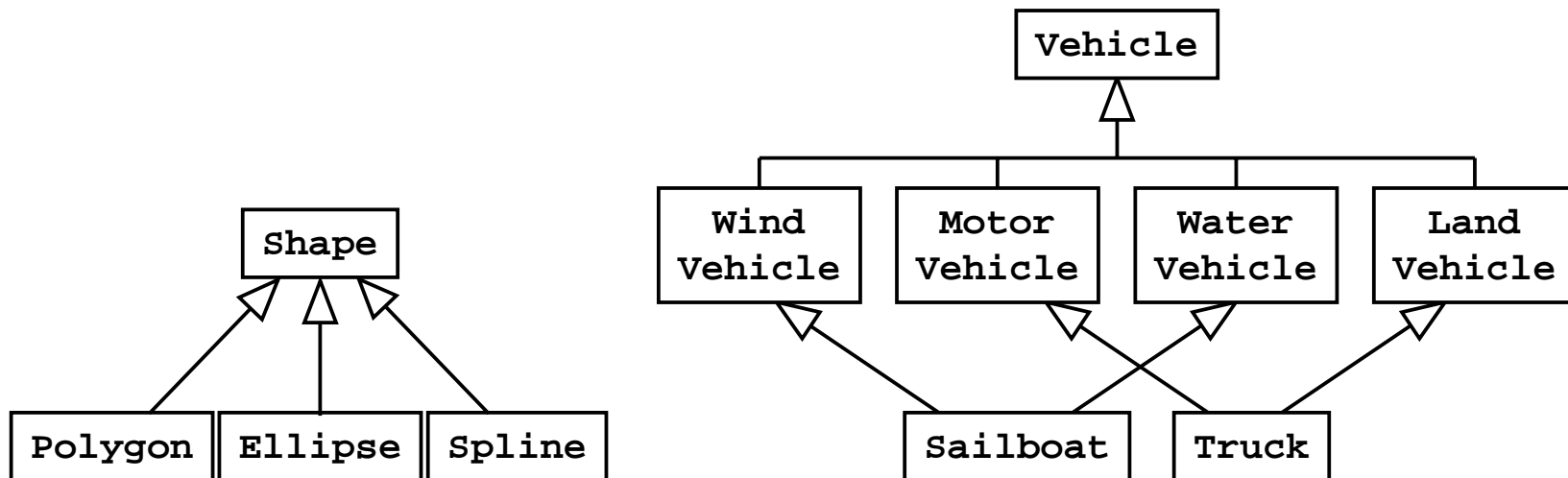
→ Defines a particular set of generalization relationships that describe the way in which a general classifier (or superclass) may be divided using specific subtypes.

❑ Usually, a generalization set describes a particular aspect of specialization.

❑ Covering and disjoint properties of a generalization set:

- {complete, disjoint} - Indicates the generalization set is covering and its specific classifiers have no common instances.

- {incomplete, disjoint} - Indicates the generalization set is not covering and its specific classifiers have no common instances.

- {complete, overlapping} - Indicates the generalization set is covering and its specific classifiers do share common instances.

- {incomplete, overlapping} - Indicates the generalization set is not covering and its specific classifiers do share common instances.

- default is {incomplete, disjoint}

❑ Generalization set may define the *powertype* - a (meta)class whose instances are subclasses of another class.

Possible Notations of Generalization Sets
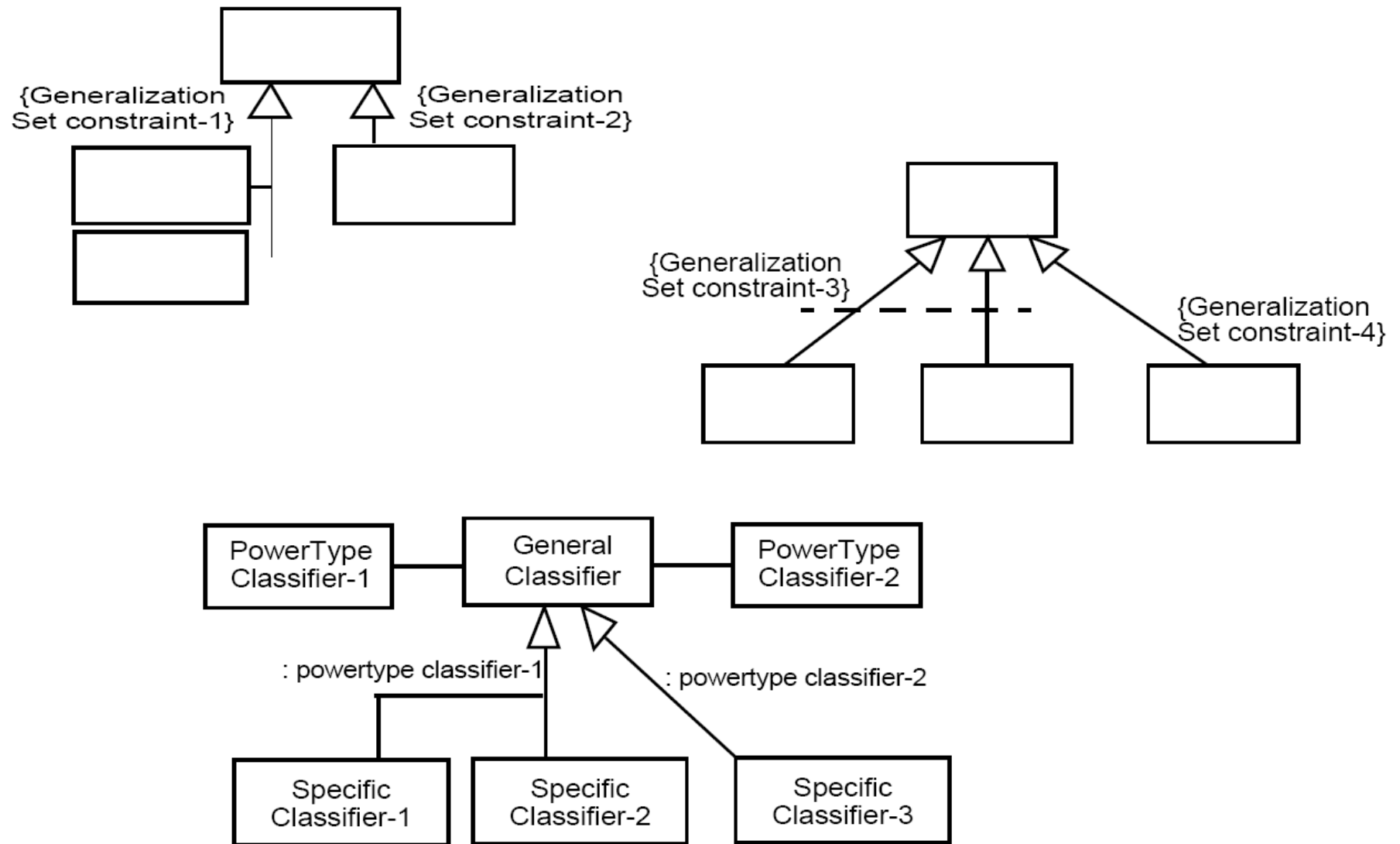
Examples of Generalization Sets (1)

Examples of Generalization Sets (3)

Navigation Expressions (OCL)
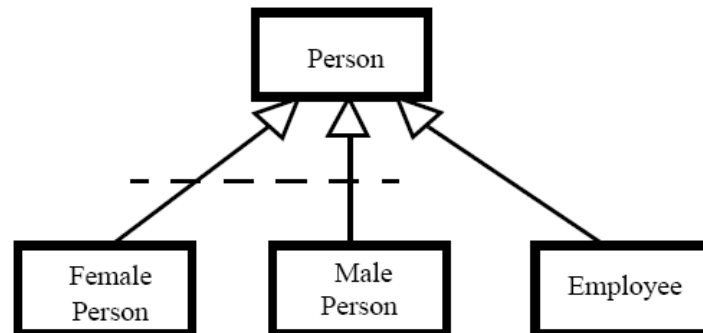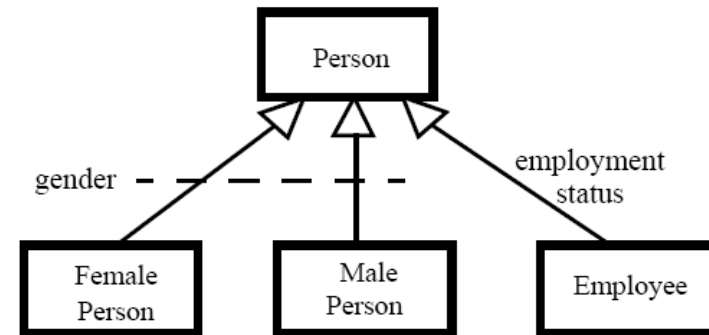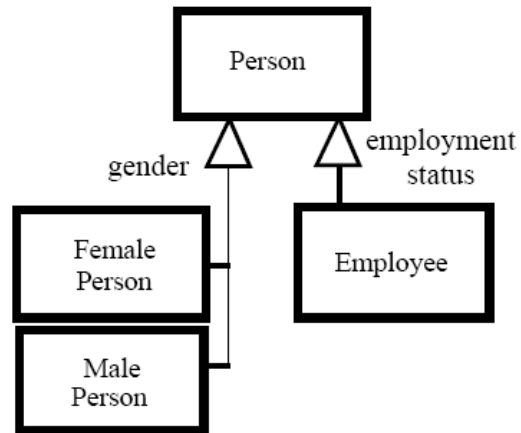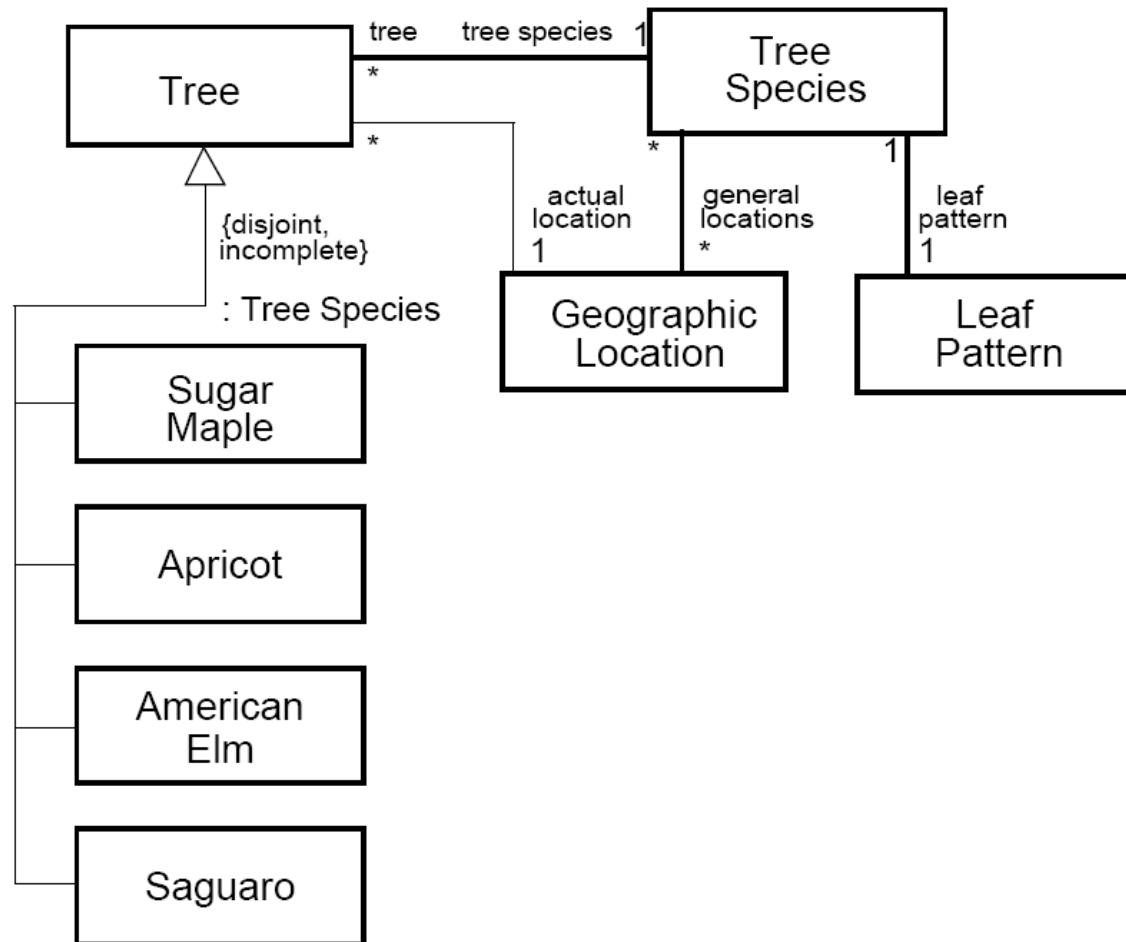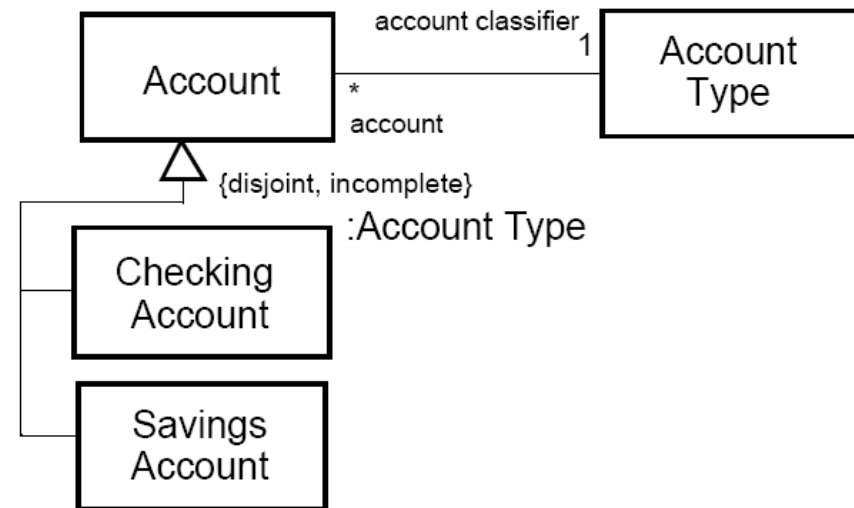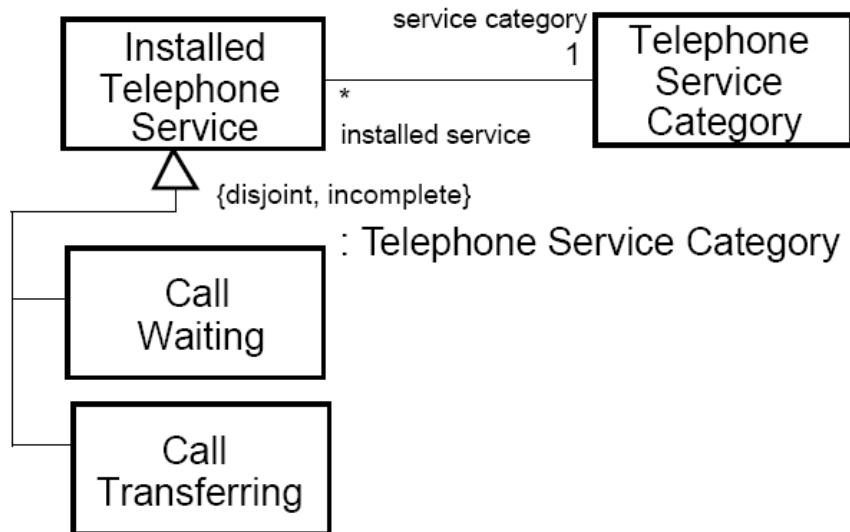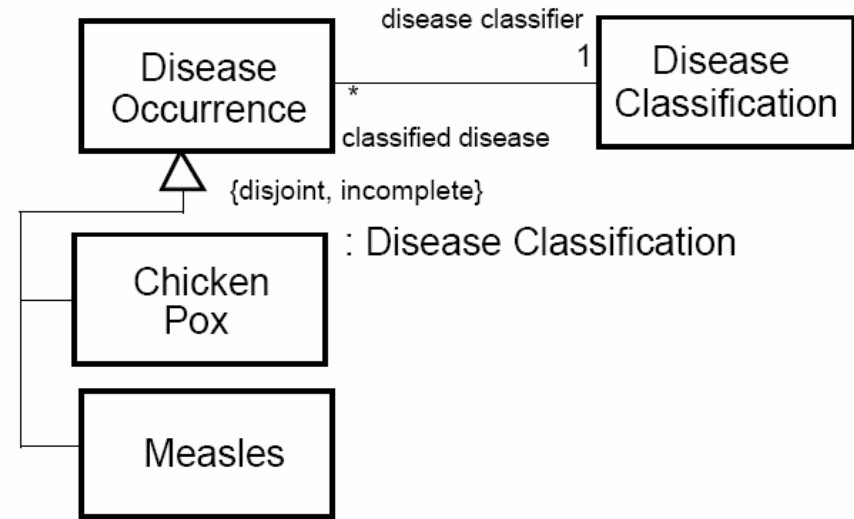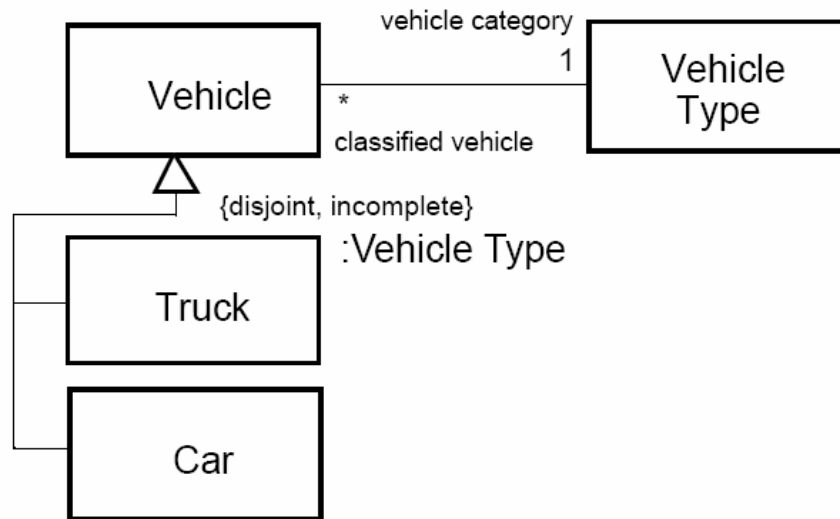
→ Allow to express navigation in models.

❐ *item.selector*

- The *selector* is the name of an attribute in the *item* or the role name of the target end of a link attached to the item. The result is the value of an attribute or related object(s).

❐ *item.selector* [*qualifier-value*]

- The *selector* designates a qualified association that qualifies the *item*. The *qualifier-value* is a value for the qualifier attribute. The result is related object selected by the qualifier.

❐ *set->*select(*boolean-expression*)

- The *boolean-expression* is written in terms of objects within the *set*. The result is the subset of objects in the set for which the *boolean-expression* is true.

Examples of Model Navigation (1)

```
                    ┌─────────────────┐
                    │      Bank       │
                    ├─────────────────┤
                    │ account: Integer│
                    └────────┬────────┘
                           0..1
                           │
                      customer
  ①  ┌──────────────────┐                              ┌──────────────────────┐
     │      Person      │          ②                   │       Company        │
     ├──────────────────┤ ─────────────────▶           ├──────────────────────┤
──▶  │ name:String      │ employee         employer    │ name:String          │
     │ age:Integer      │ 0..*                  0..*    │ residence:String     │
     └──────────────────┘                              └──────────────────────┘
```
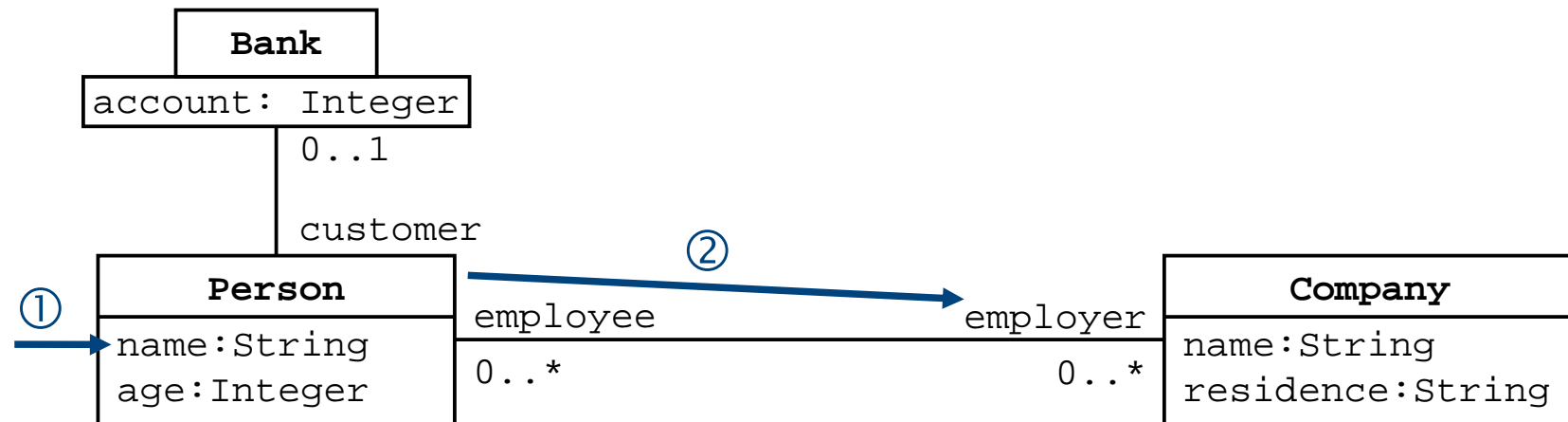
①   Name of a person:
Person.name

②   Names of person's employers:
Person.employer.name

Examples of Model Navigation (2)
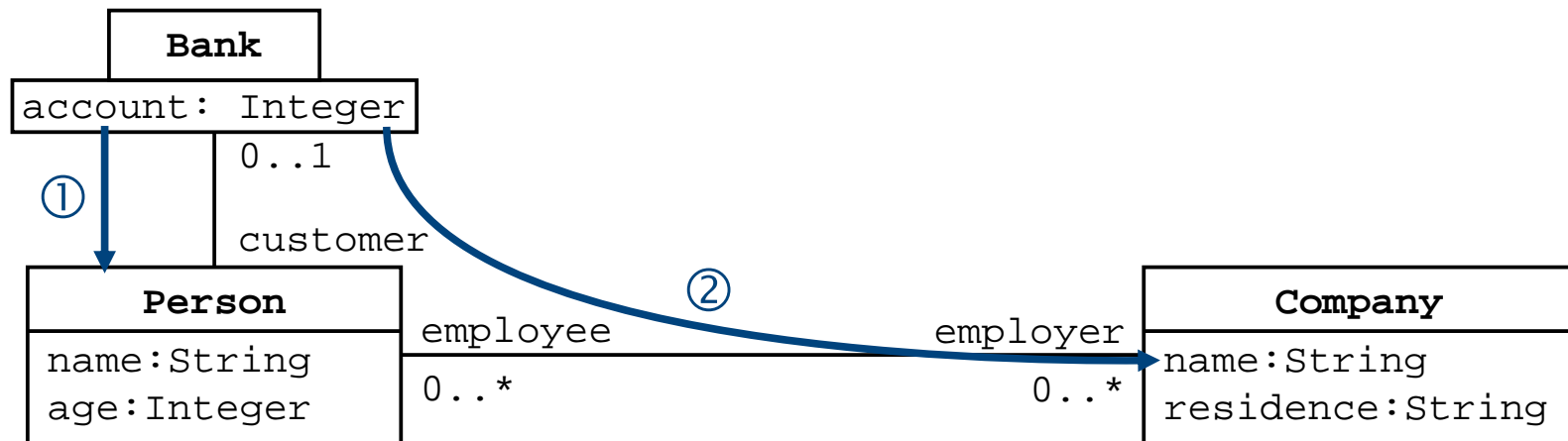
① A customer of the bank with the account num. 8526:
   Bank.customer[8526]

② Employers of an owner of the account 6251:
   Bank.customer[6251].employer.name
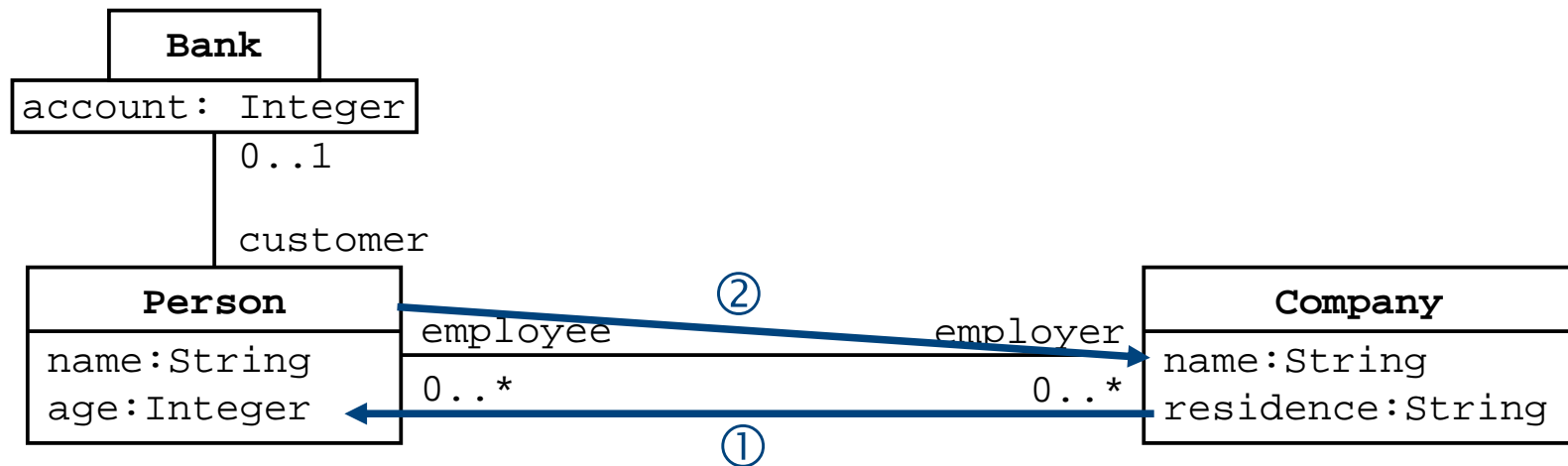
① Employees older than 50:
Company.employee -> select(p|p.age>50)

② Names of employers from Bratislava:
Person.employer ->
select(c|c.residence='Bratislava').name

Instance Specification

→ Representation of an instance in a modeled system.

❑ Can specify name and one or more classifiers:

[*name*] ':' [*classifier-name* [',' *classifier-name*]*]

❑ Kind of the instance specification is given by its classifier(s). It can be:

- Object
  - An instance of a class.
  - Can specify values of structural features of the entity–slots:

    [[*name*] [':' *type*] '='] *value*

- Link
  - A tuple (mostly a pair) of object references.
  - An instance of an association.
  - Association adornments can be shown, except of multiplicity.
- etc.

❑ Visually, the instance specification shares the shape of its classifier(s).

Examples of Instance Specifications (1)

```
┌─────────────────────────┐
│   streetName:String     │
│  "Baker Street 21b"     │
└─────────────────────────┘
```

```
┌───────────────────────────────────┐
│     holmesAddress:Address         │
├───────────────────────────────────┤
│ streetName="Baker Street"         │
│ streetNumber="21b"                │
└───────────────────────────────────┘
```

```
┌──────────────┐
│   triangle   │
└──────────────┘
```

```
┌──────────────┐
│  :Polygon    │
└──────────────┘
```

```
┌────────────────────┐
│  triangle:Polygon  │
└────────────────────┘
```

```
┌───────────────────────────────────┐
│        triangle:Polygon           │
├───────────────────────────────────┤
│ center=(0,0)                      │
│ vertices=((0,0),(4,0),(4,3))      │
│ borderColor=black                 │
│ fillColor=white                   │
└───────────────────────────────────┘
```

```
                    :Fathership
┌────────────┐                      ┌──────────────┐
│ Don:Person │──────────────────────│ Josh:Person  │
└────────────┘  father        son   └──────────────┘
```

officer   **Baca:Person**

santaFeRFC:RugbyClub
- treasurer
- president

member

member   **Podmore:Person**

member

officer   **Morris:Person**

Process of Class Modeling

1. Identify classes

   - From glossary.

   - From a business model or business-related artifacts.

   - From the stored information items and business artifacts.

   - From use case realizations.

2. Specify the semantics of classes

   - Responsibility.

   - Attributes, operations and interfaces.

3. Identify relationship among classes

   - Domain-based associations.

   - From object interactions.

   - Generalization and aggregation relationships.

4. Structure the model into packages

5. Repeat the process and refine the model.

Object-Oriented Analysis and Modeling

UML Composite Structures

Composite Structures

→ **Common mechanisms used for modeling internal structures of classifiers, interaction points isolating classifiers from their environments, classifier interfaces and collaborations.**

❒ The term "structure" refers to a composition of interconnected elements, representing run-time instances collaborating over communications links to achieve some common objectives.

Applied for:

- Classes.

- Components.

- Collaborations.

- Nodes.

Diagrams:

- Composite structure diagram.

- All composite structures can also be shown on other structure diagrams.

Property Owned as a Part

→ Represents a set of instances that are owned by a containing classifier (called *structured classifier*) instance.

❑ Containment by composition.

❑ When an instance of the containing classifier is created, a set of instances corresponding to its properties may be created either immediately or later.

❑ All such instances are destroyed when the containing classifier instance is destroyed.

❑ Format: {*name* ['/' *rolename*] | '/' *rolename*}
['*:*' *classifiername* ['*,*' *classifiername*]*]

❑ A specialized *connectable element*.

ContainingClassifier			
part1:C1	part2:C2[*]	part3 [4]	part4:C4[2]
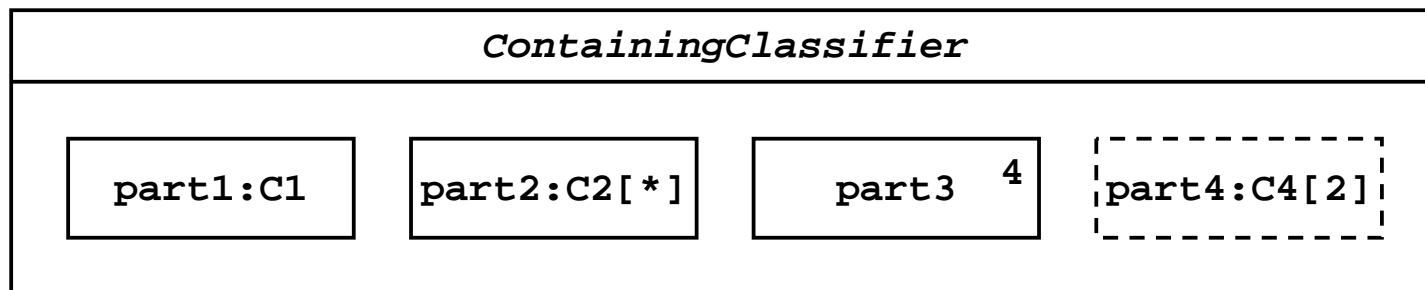
Examples of Parts
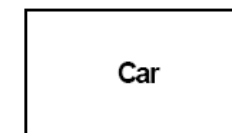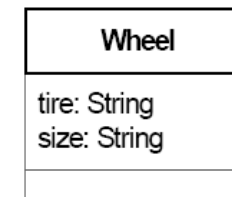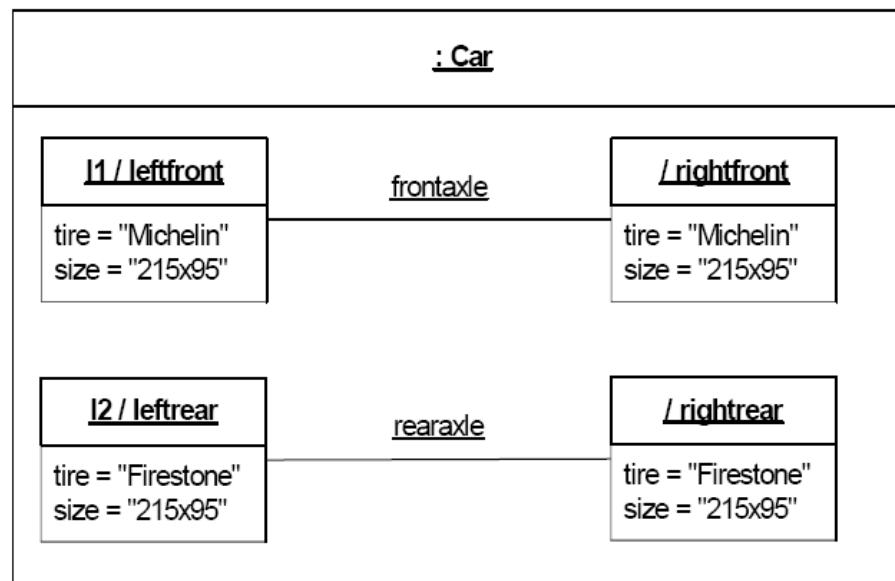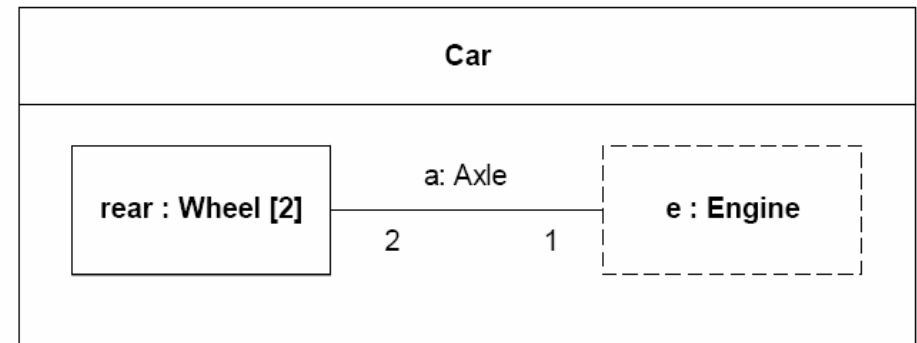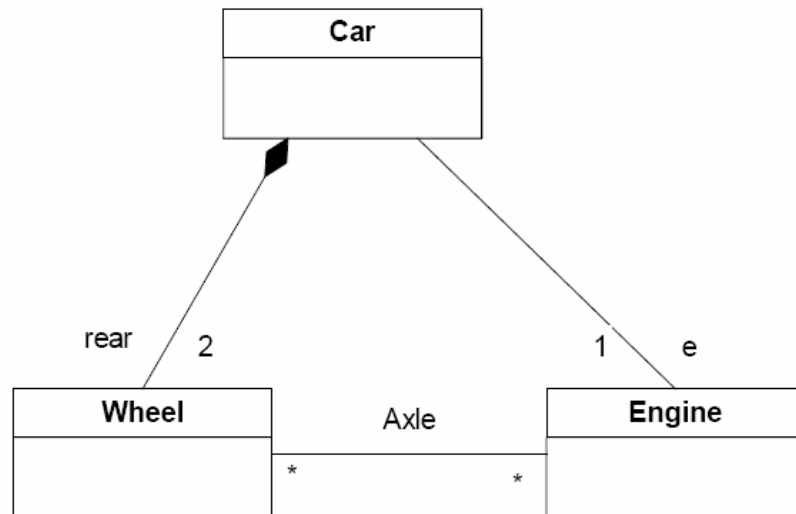
Port

→ A property of a classifier (called *encapsulated classifier*) that specifies a distinct **interaction point** between that classifier and its environment or between the (behavior of the) classifier and its internal parts.

❒ Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier.

❒ A port may specify the services (as associated interfaces) a classifier *provides* to its environment as well as the services that a classifier *expects* of its environment.

- The required interfaces characterize the requests that may be made from the classifier to its environment through this port.

- The provided interfaces characterize requests to the classifier that its environment may make through this port.

❒ Type of a port must realize provided interfaces.

❒ A port by default has public visibility.

❒ A specialized *property* (and therefore also *connectable element*).
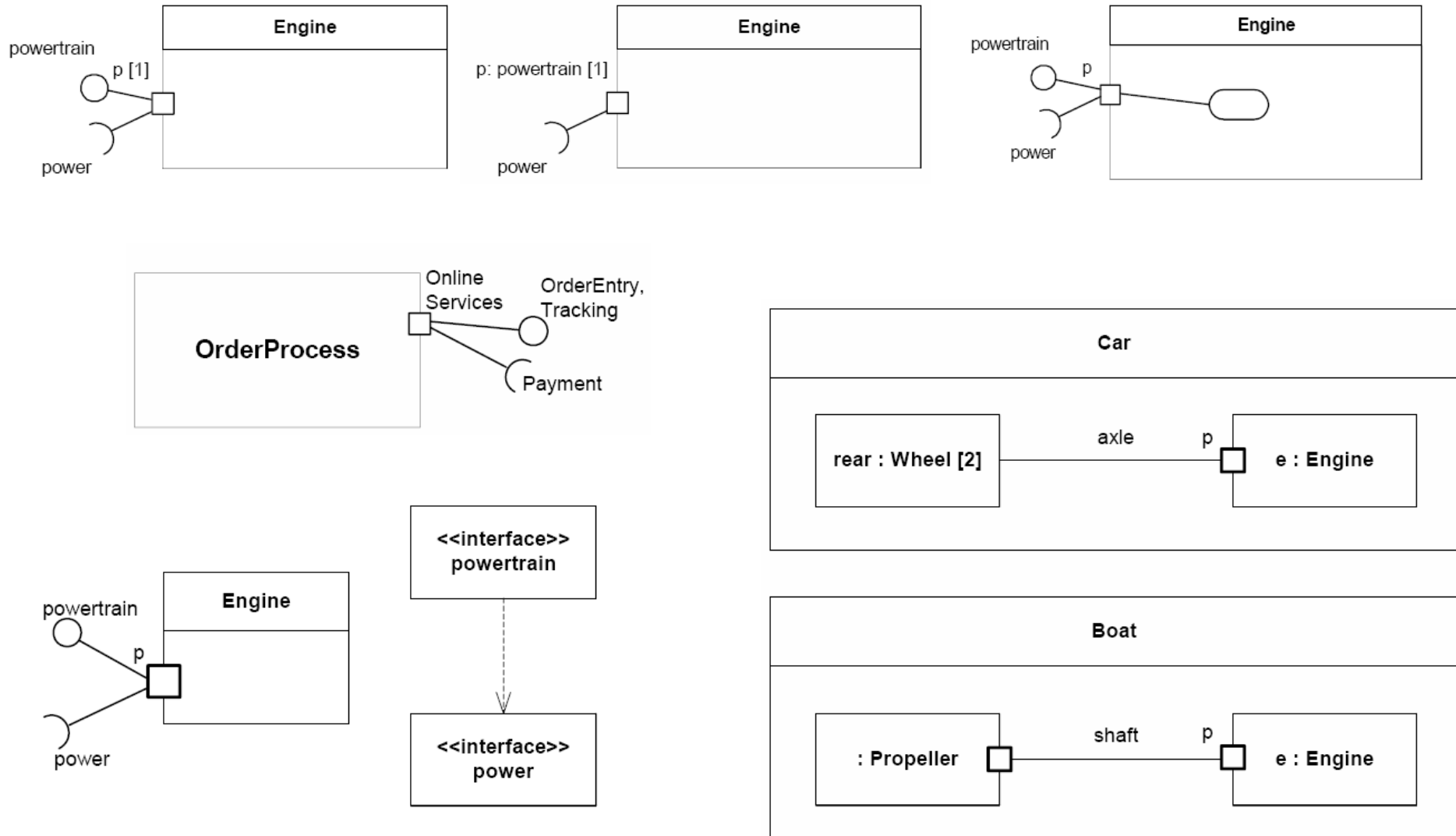
Connector

→ A link that enables communication between two or more instances represented by connectable elements.

❒ May be an instance of an association, or a link between parameters, variables, slots, or the same instance.

❒ May be realized by a "simple" pointer or by something as "complex" as a network connection.

❒ Syntax: ([*name*] ':' *association*) | *name*

❒ Two or more connector ends

  ● Adornments as for association ends.

  ● No multiplicity ⇒ matches the multiplicity of the end it is attached to.

Collaboration

→ Description of a structure of collaborating elements (their roles) to be played by instances, each performing a specialized function, which collectively accomplish some desired functionality.

❑ Cooperating entities are the properties of the collaboration.

❑ Connectors define communication paths between the participating instances.

❑ Can own behavior(s), e.g., interaction(s) of roles.

❑ Incorporates only aspects that are necessary and suppresses everything else, e.g., the precise class of participating instances, or their irrelevant features and links.

❑ A specialized *structured classifier* and *behaviored classifier*.

Collaboration
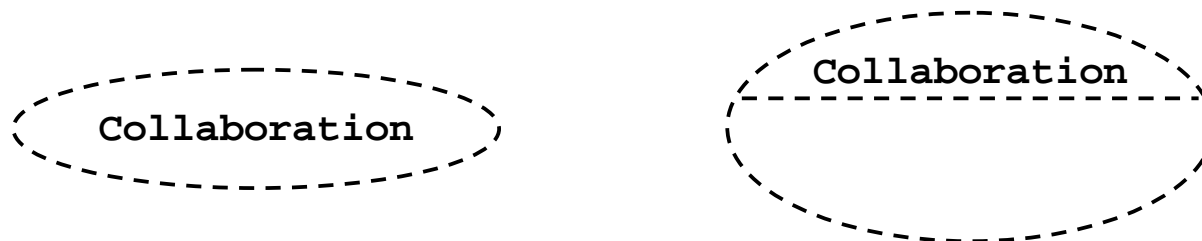
Collaboration

Examples of Collaborations
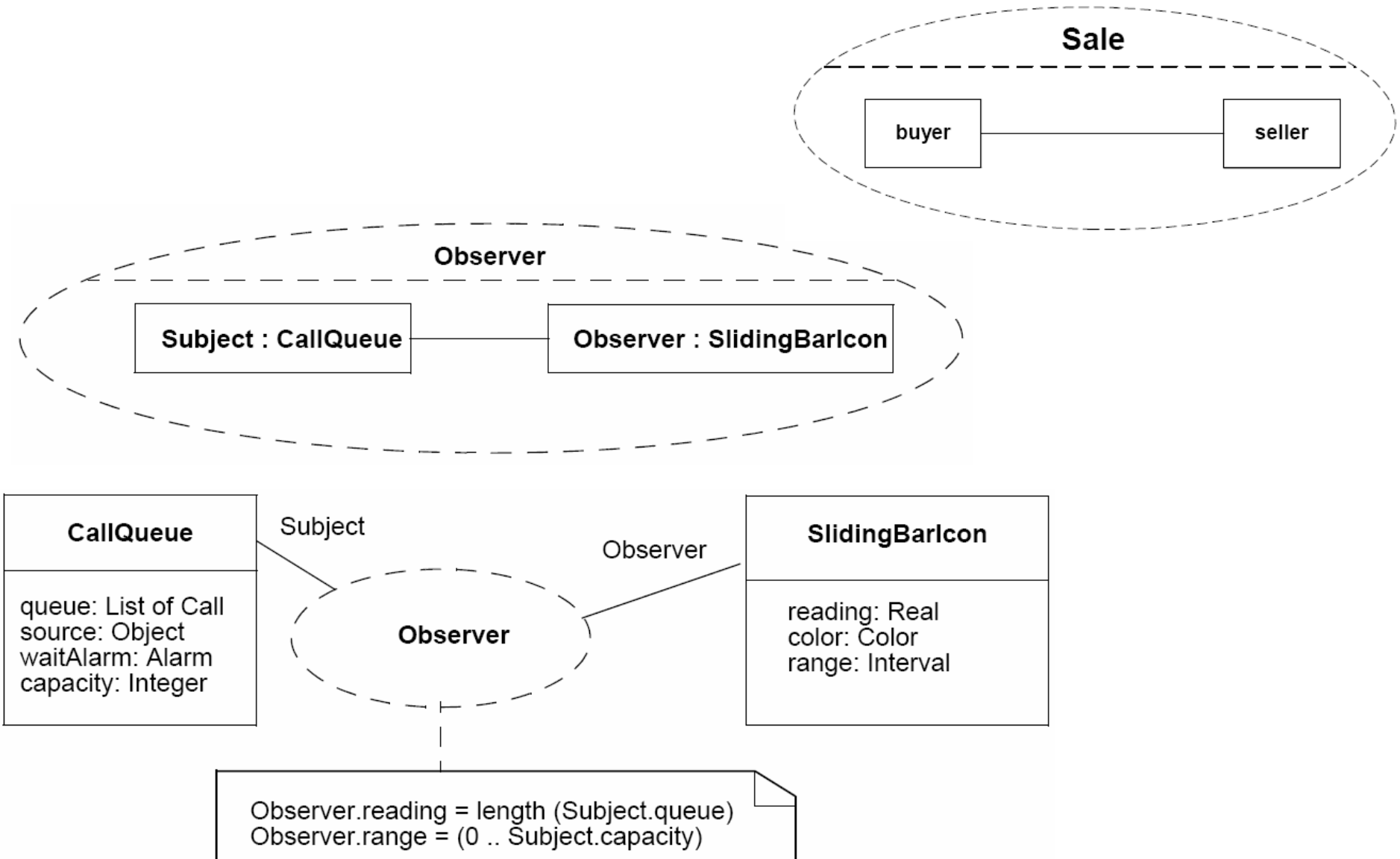
Collaboration Use

→ The application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration.

❏ A collaboration use relates a feature in its collaboration type to a connectable element in the classifier or operation that owns the collaboration use.

❏ Any behavior attached to the collaboration type applies to the set of roles and connectors bound within a given collaboration use.
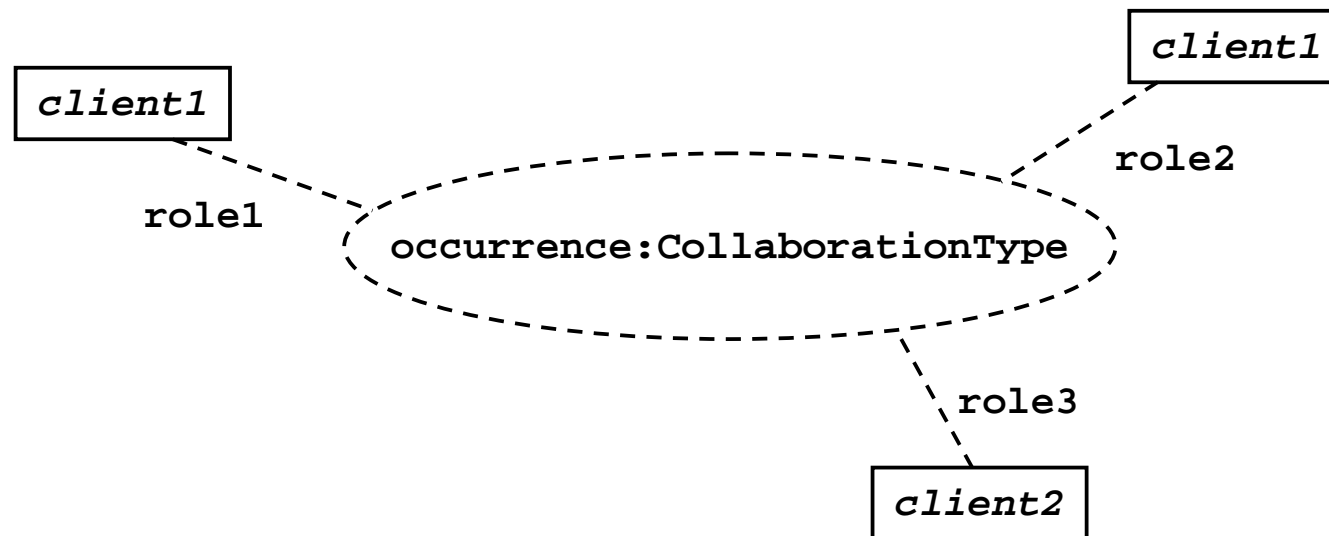
Example of Collaboration Use
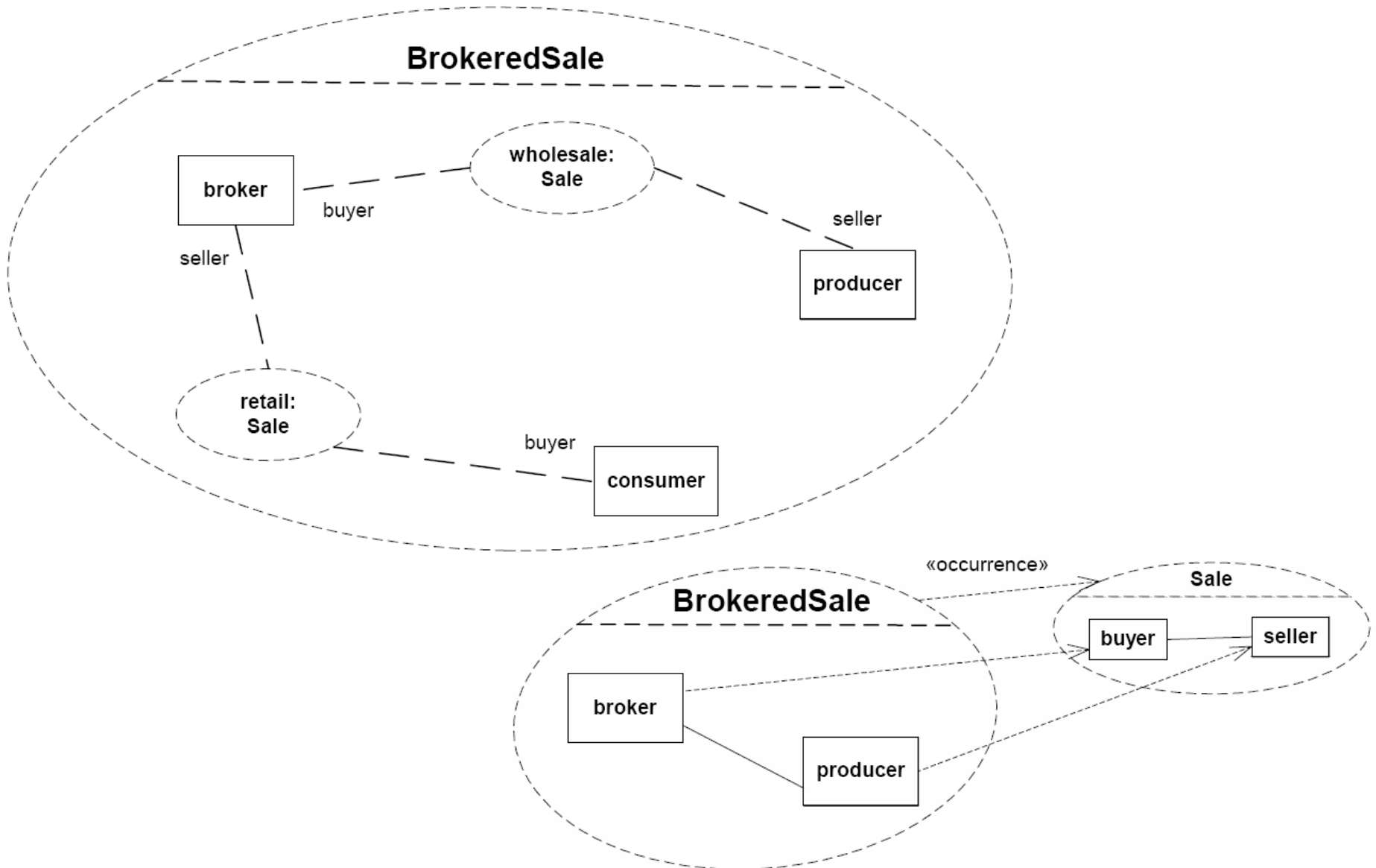
Object-Oriented Analysis and Modeling

Components

Component Model

→ **Decomposition of the system into reusable, modular, executable logical or physical units—components.**

Consists of:

- Component diagrams.

- Element descriptions.

Used (mainly) in:

- Design $\Rightarrow$ logical and physical component structuring of the system.

Component

→ A self contained part of a system that encapsulates its contents (state and behavior of a number of classifiers) and whose manifestation is replaceable within its environment.

❒ Specifies a formal contract of the services that it <u>provides</u> to its clients and those that it <u>requires</u> from other components or services in the system in terms of its provided and required <u>interfaces</u>. ("external view")

- A component serves as a type whose conformance is defined by these provided and required interfaces.

- One component may therefore be substituted (at design time or run-time) by another only if the two are type conformant.

❒ A specialized Class. ("internal view")

⇒ Attributes, operations, owned behavior, internal structure, ports, participation in associations and generalizations.

❒ Can own and import members; as a package. ("internal view")

❒ Represents either a <u>logical</u> structure of the model or <u>physical</u> structuring of code (modules, libraries, executables, etc.).

<<component>>
ComponentName

ComponentName

Examples of Components (1)
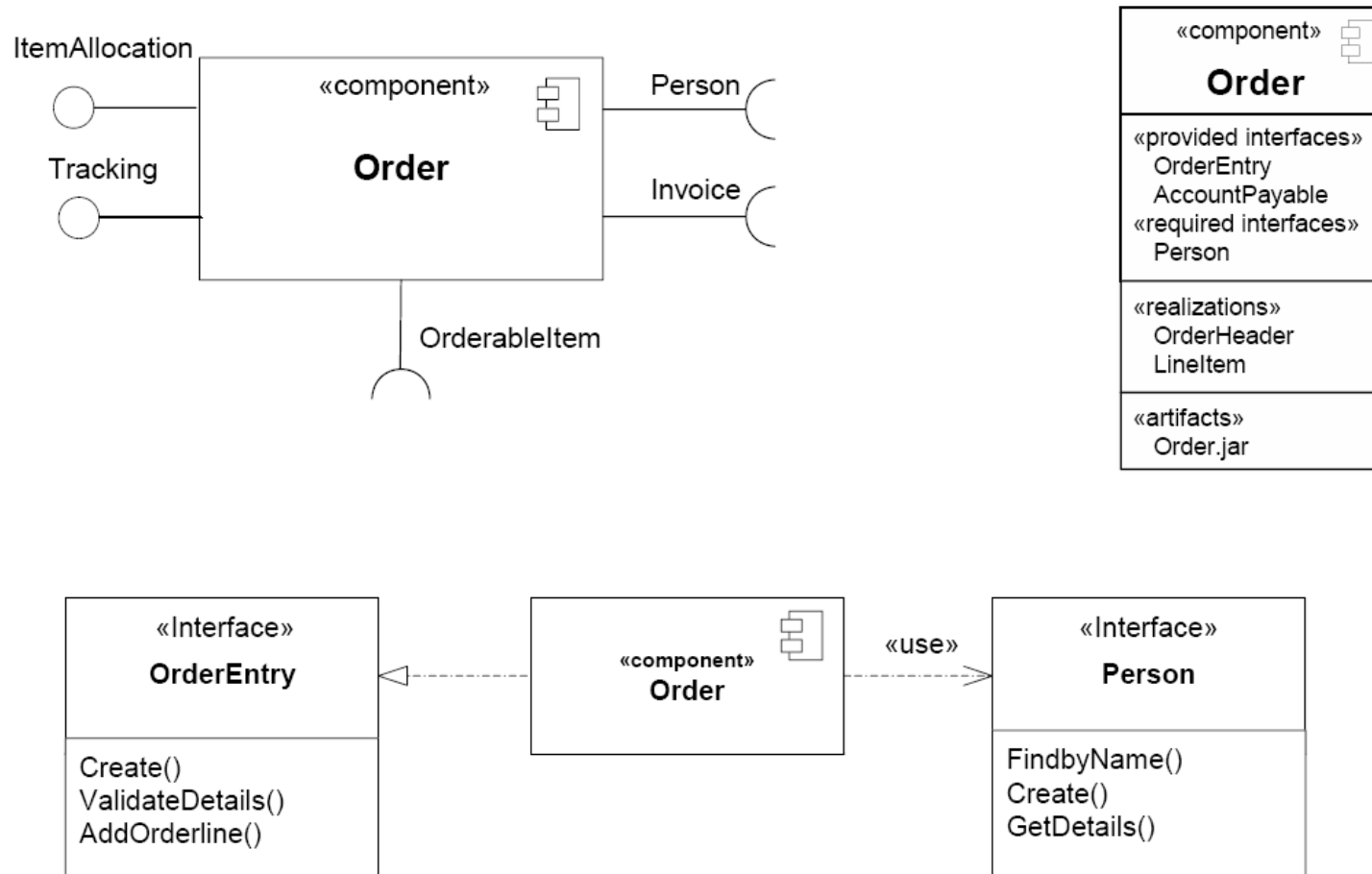
Examples of Components (2)
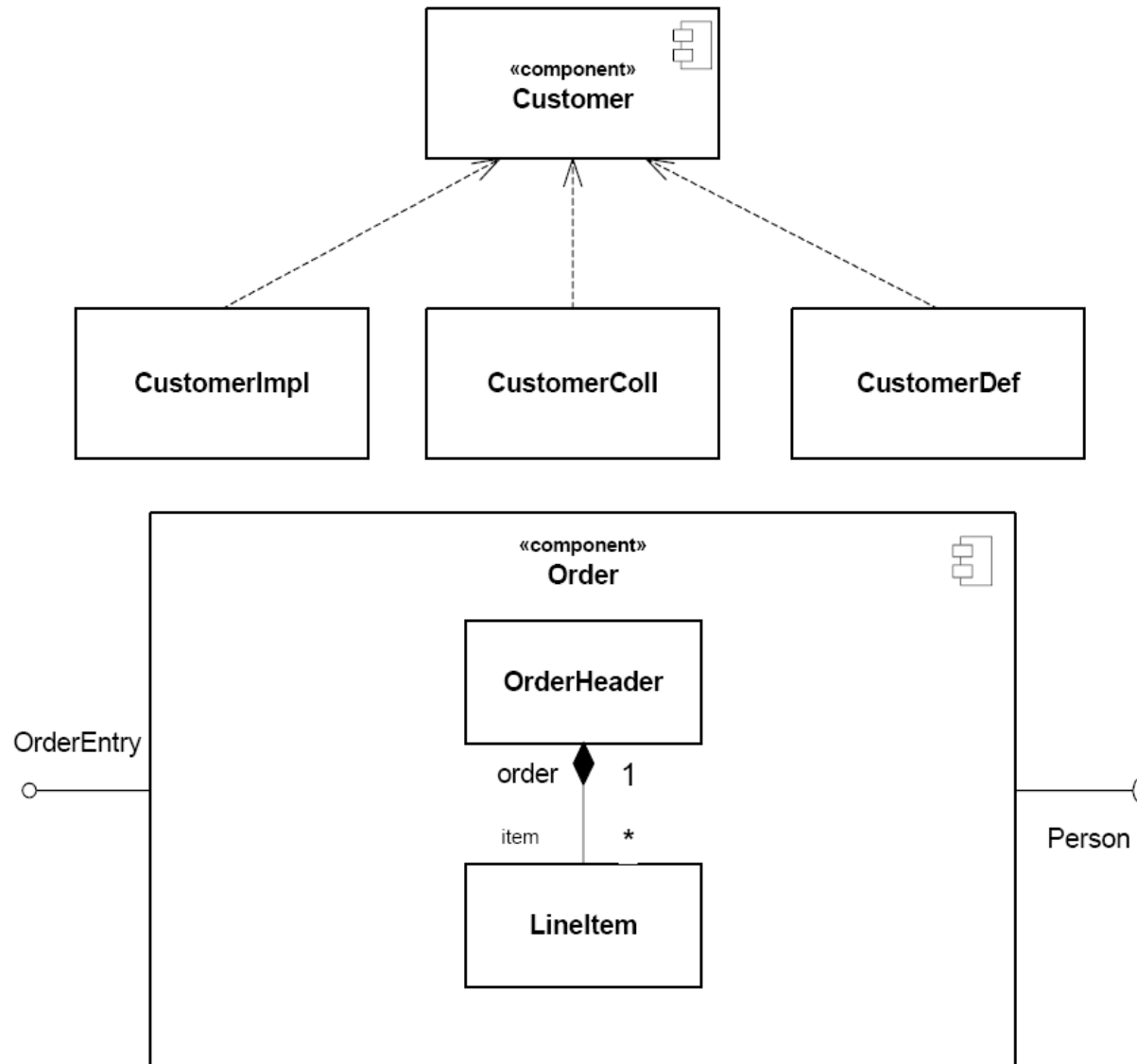
Examples of Components (3)
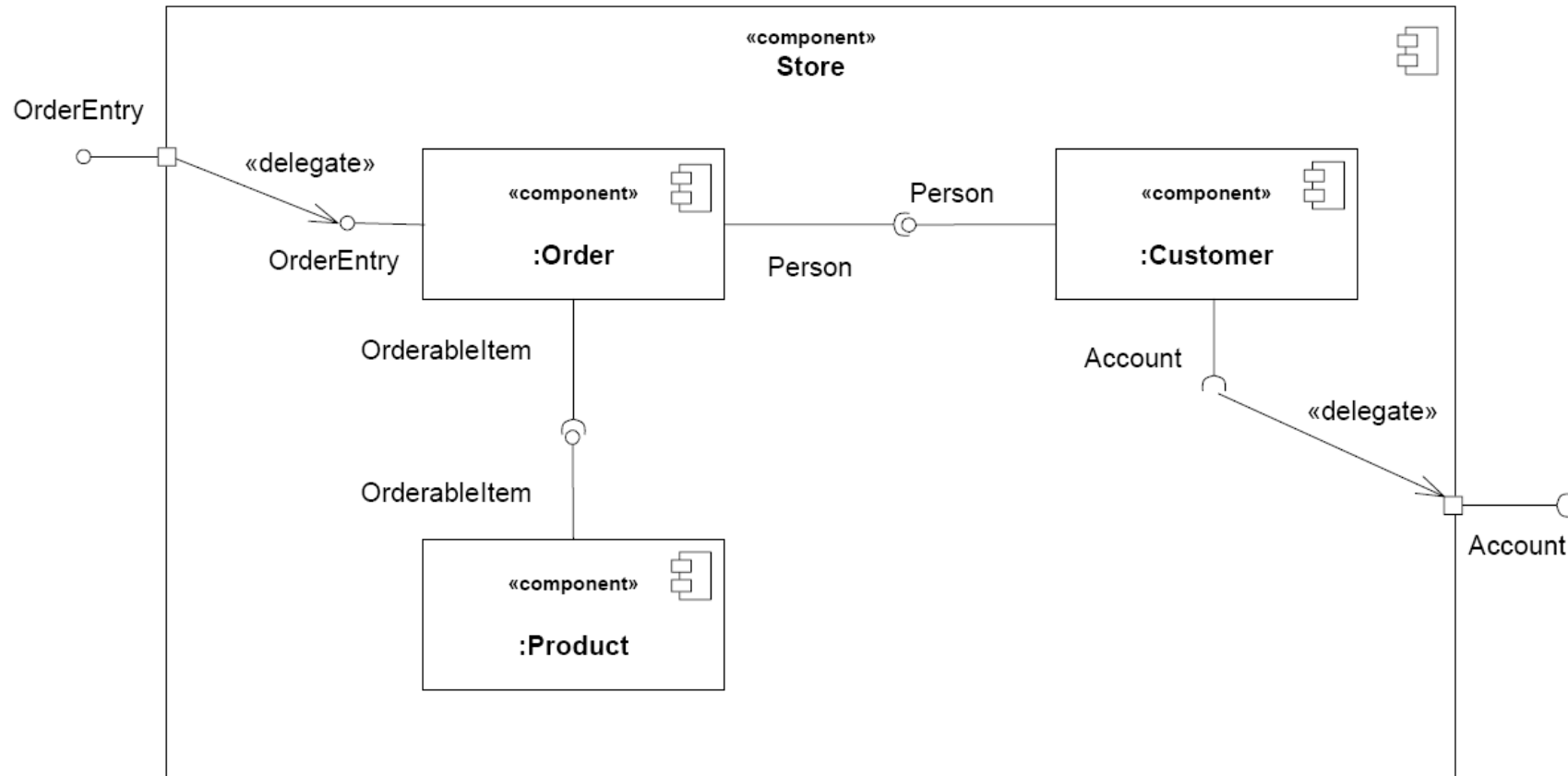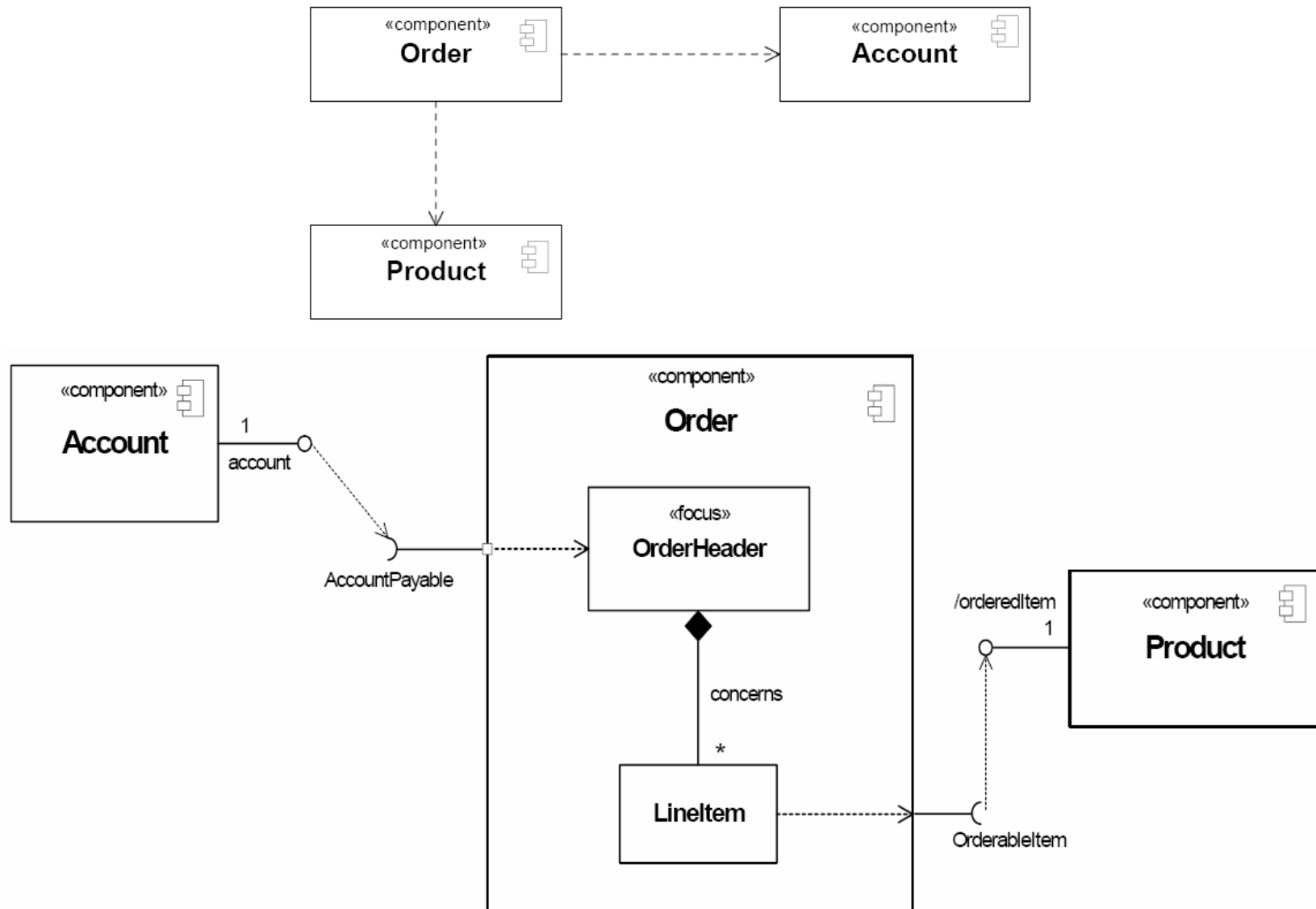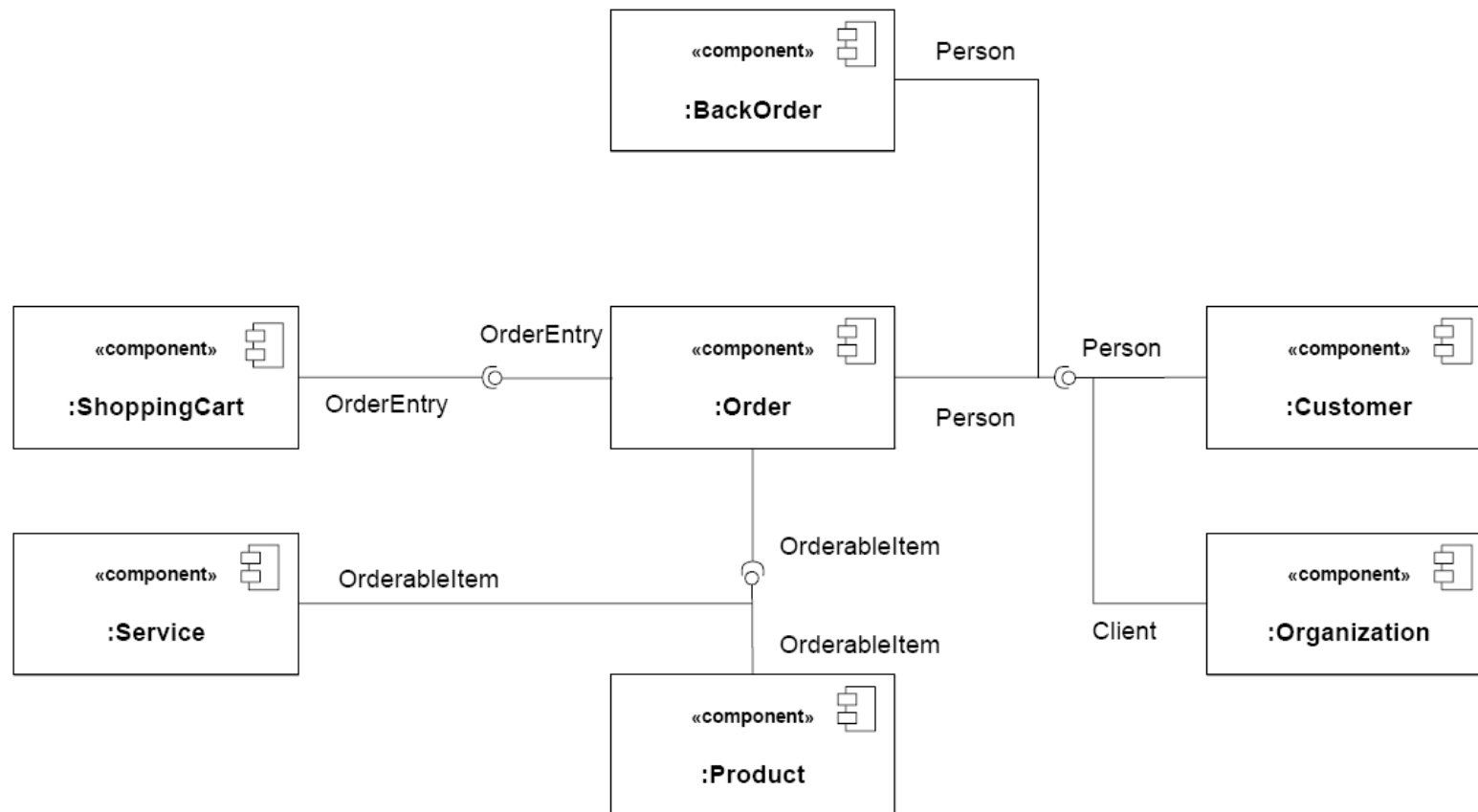
Connector

- The connector concept is extended to include interface based constraints and notation.

- A *delegation connector* is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts. It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling.

- An *assembly connector* is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

Examples of Connectors (1)
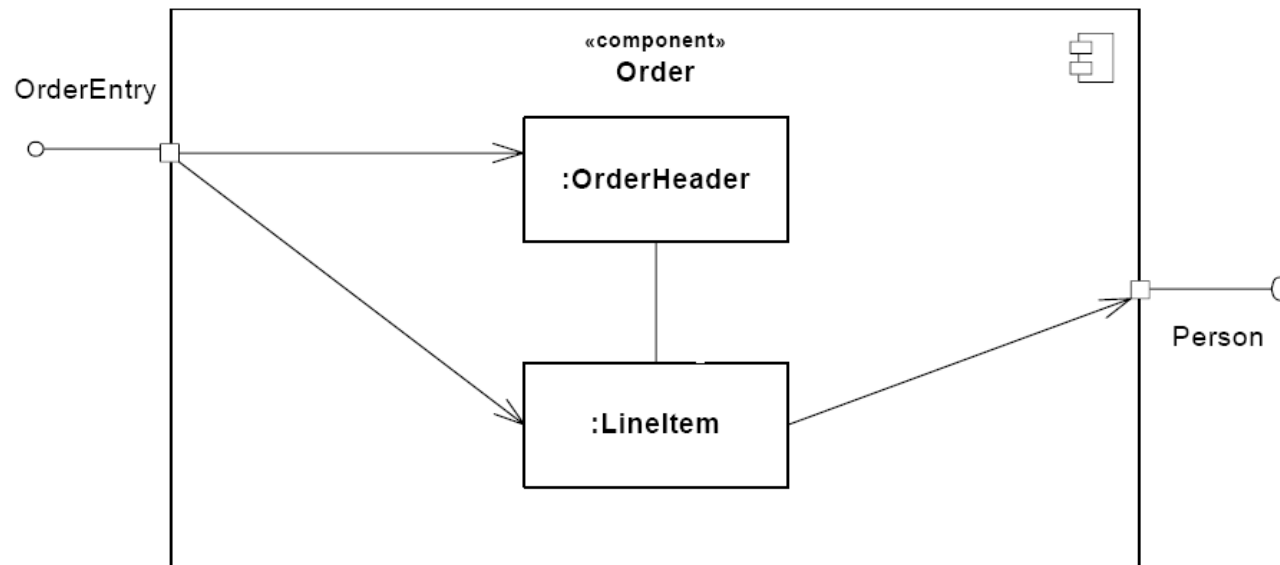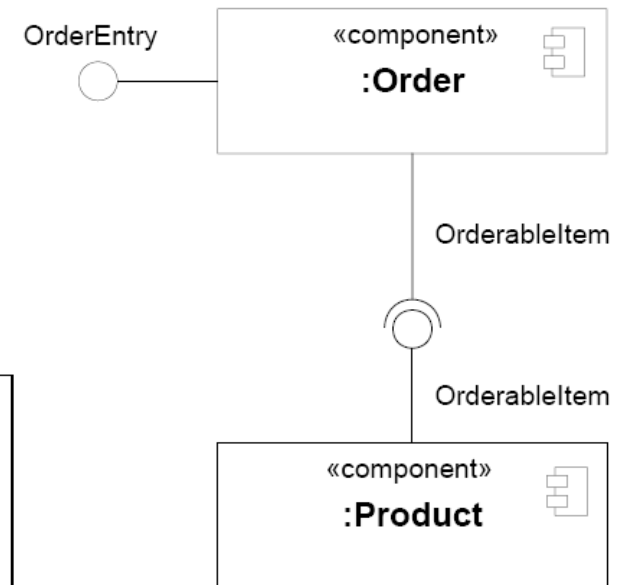
Examples of Connectors (2)
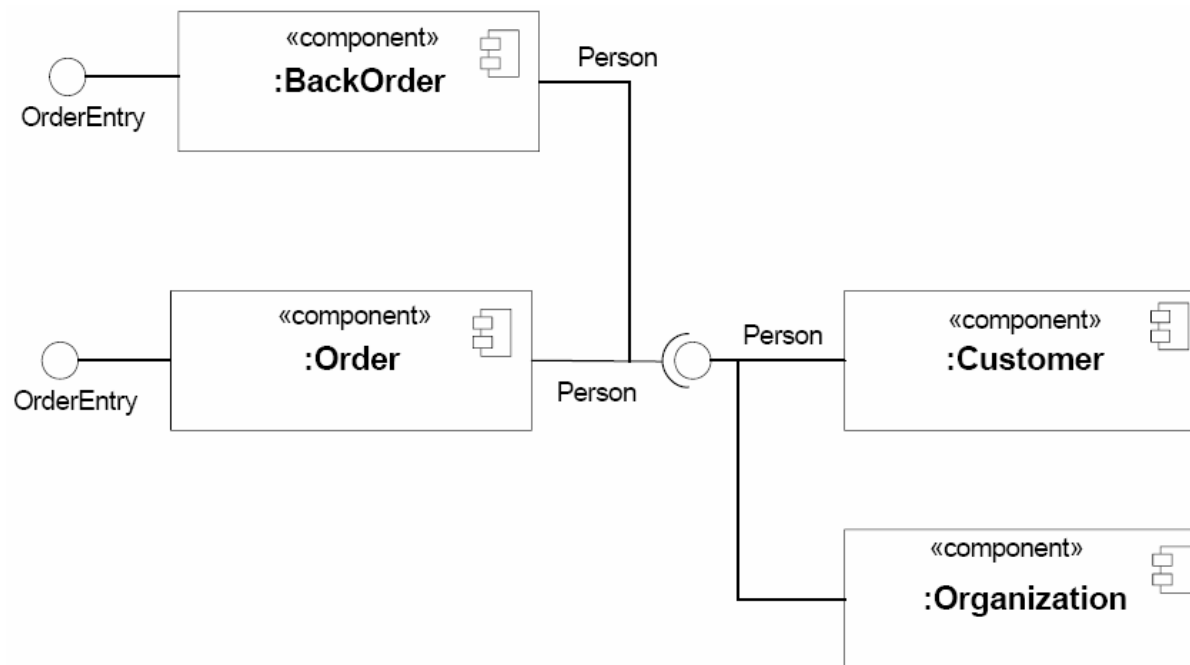
WHITESTEIN
Technologies

Object-Oriented Analysis and Modeling

Deployments

Deployment Model

→ **Defines the execution architecture of system that represent the assignment of software artifacts to nodes, plus internal and external structure of nodes.**

Consists of:

- Deployment diagrams.

- Element descriptions.

Used (mainly) in:

- Design $\Rightarrow$ physical deployment of the system.

Node

→ Computational resource upon which artifacts may be deployed for execution.

❑ Nodes can be interconnected through communication paths to define network structures.

❑ A node can be nested in another node as an owned member.

❑ A specialized class.

❑ Therefore, nodes may have an internal structure defined in terms of parts and connectors.

Name

name:Type
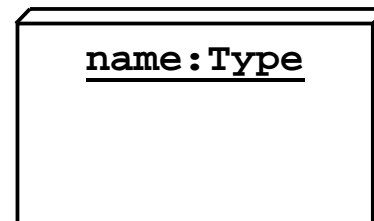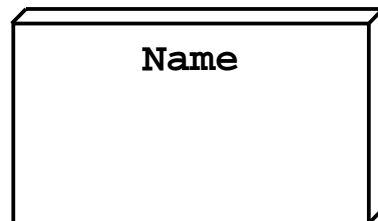
Device

→ A physical computational resource with processing capability upon which artifacts may be deployed for execution.

❐ Devices may be complex, i.e., they may consist of other devices.

- This is done either through namespace ownership or through attributes that are typed by Devices.

❐ A specialized node.

```
      <<device>>
        Name
```

```
      <<device>>
      name:Type
```
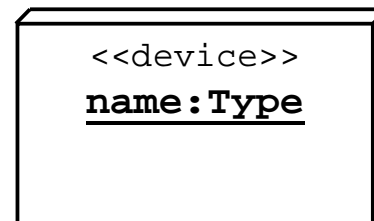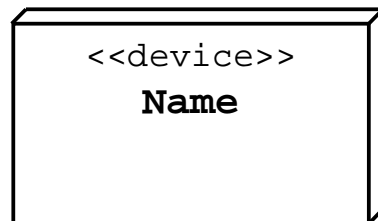
Execution Environment

→ A node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts.

❑ An execution environment usually implements a standard set of services that the deployed artifacts require at execution time (at the modeling level these services are usually implicit).

❑ Usually, execution environments are parts of other nodes (e.g., devices).

❑ Execution environments can be nested, e.g., a database execution environment may be nested in an operating system execution environment.

❑ An execution environment can optionally have an explicit interface which can be called by the deployed elements.

❑ A specialized node.

```
<<executionEnvironment>>
        Name
```

```
<<executionEnvironment>>
       name:Type
```
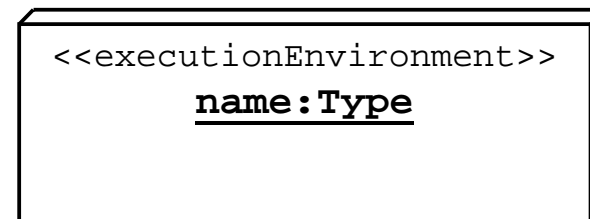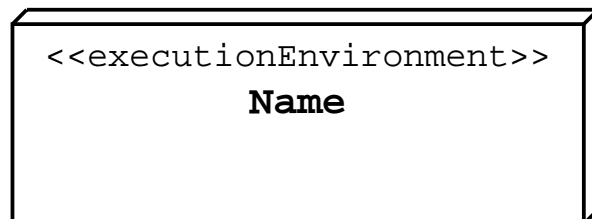
Examples of Nodes
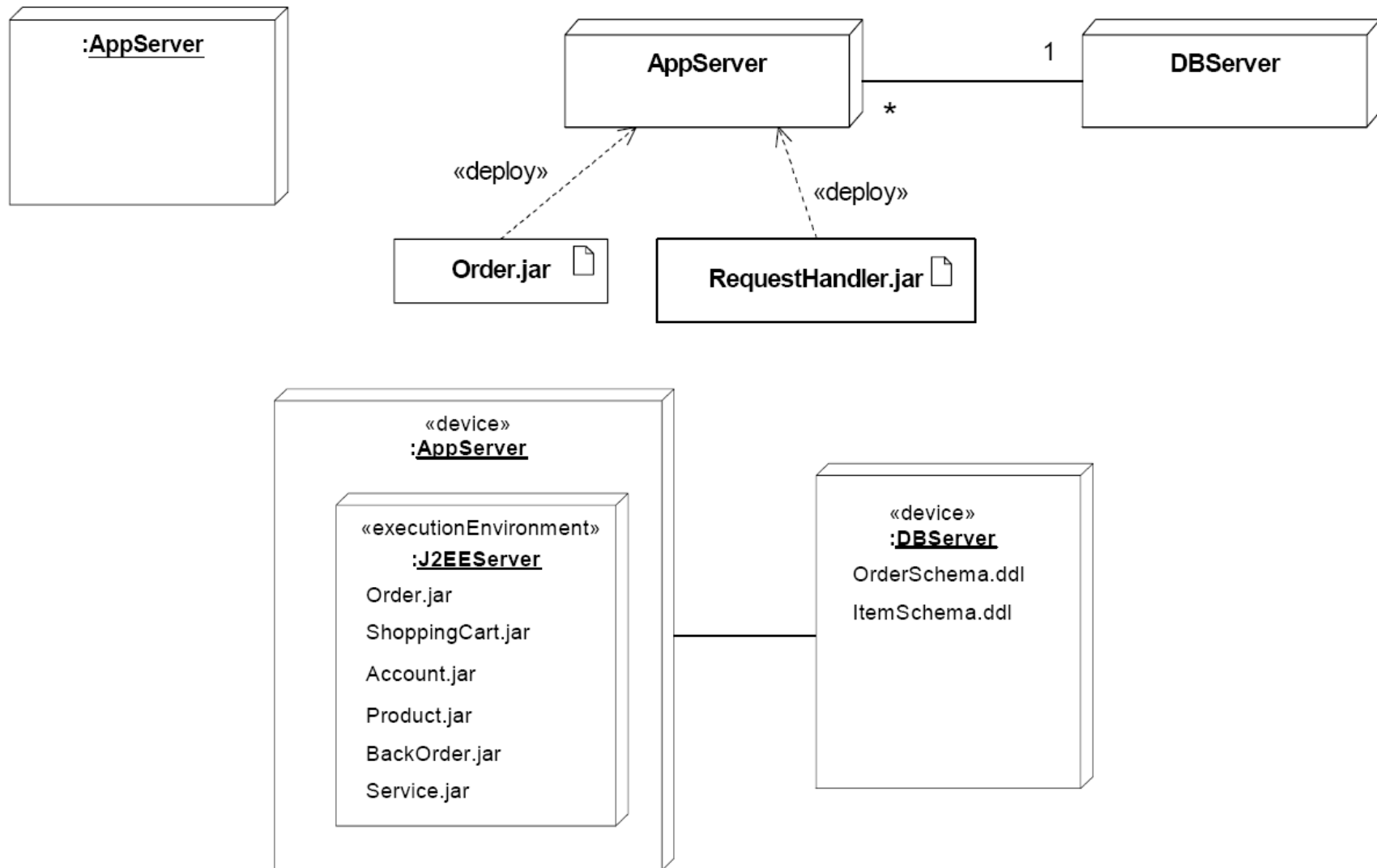
Communication Path

→ An association between two deployment targets, through which they are able to exchange signals and messages.

❏ A specialized association.

❏ No special notation.

Artifact

→ The specification of a physical piece of information (a concrete element in the physical world) that is used or produced by a software development process, or by deployment and operation of a system.

❑ Examples of artifacts include model files, source files, scripts, binary executable files, configuration files, a table in a database system, a development deliverable, or a word-processing document, a mail message.

❑ Can be deployed to nodes (by means of the deployment relationships).

❑ Can own operations and attributes, and can be associated with other artifacts.

❑ A specialized classifier.

```
┌─────────────────────────┐        ┌─────────────────────────┐
│   <<artifact>>     ▯     │        │   <<artifact>>     ▯     │
│      Name                │        │     name:Type            │
│                          │        │                          │
└─────────────────────────┘        └─────────────────────────┘
```
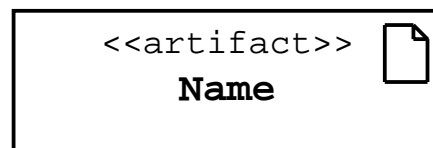
Manifestation

→ The concrete physical rendering of one or more model elements by an artifact.

❏ An artifact embodies or manifests a number of (packageable) model elements.

❏ One element can be utilized in none, one, or more manifestations.

❏ A manifestation is owned by an artifact.

❏ A specialized abstraction (dependency).

```
┌────────────────────────┐                    ┌────────────────────────┐
│  <<artifact>>     ▯     │    «manifest»      │   UtilizedElement      │
│    Artifact            │- - - - - - - - - - >│                        │
│                        │                    │                        │
└────────────────────────┘                    └────────────────────────┘
```
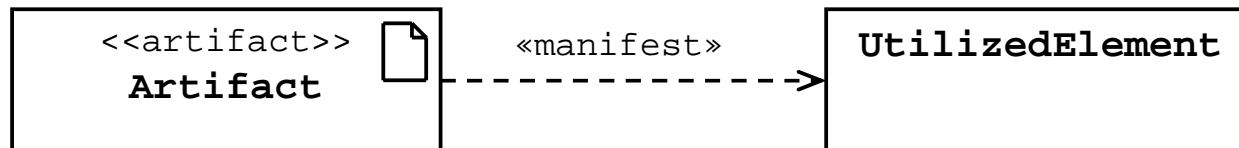
Examples of Artifacts and Manifestations
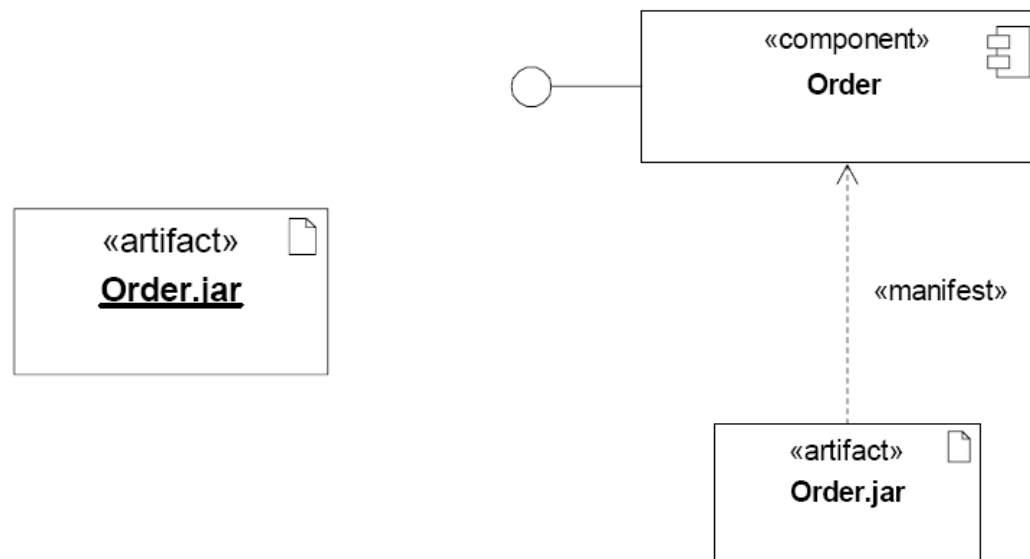
Deployment

→ A relationship used to specify the allocation of an artifact or artifact instance to a deployment target.

❑ It can optionally be parameterized by a deployment specification.

❑ Deployment target = node, property or instance specification.

❑ A specialized dependency.
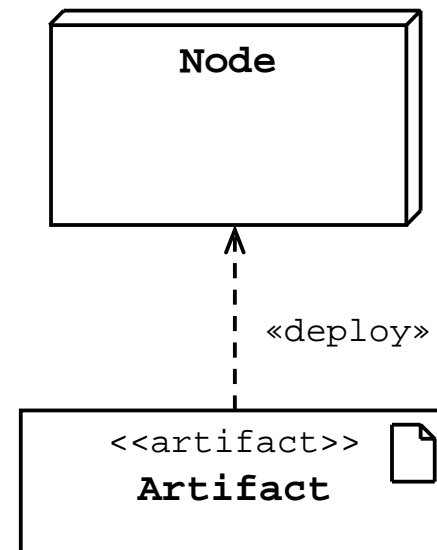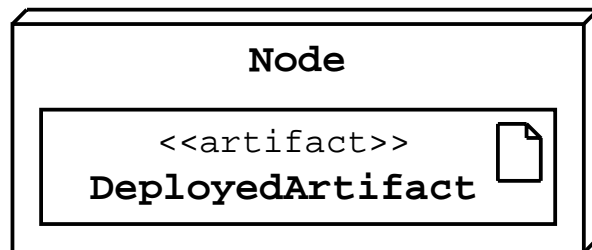
Examples of Deployments
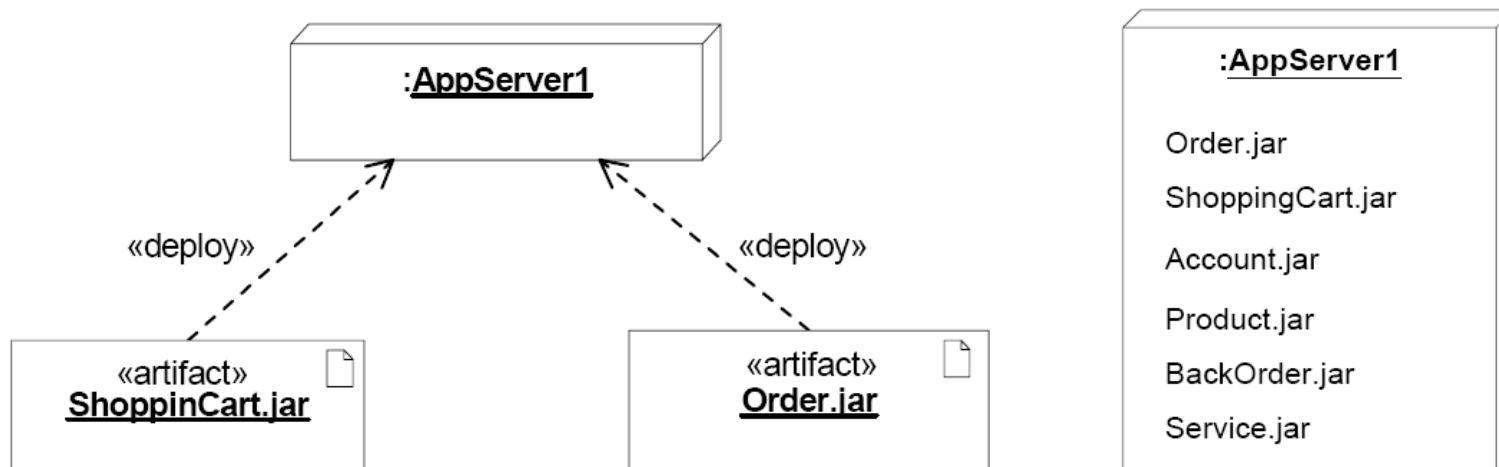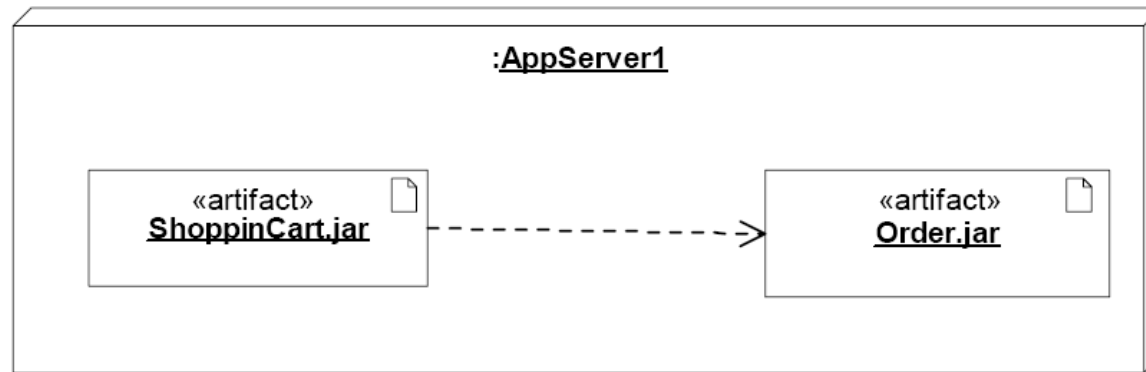
Deployment Specification

→ Specifies a set of properties that determine execution parameters of a component artifact that is deployed on a node.

❏ A deployment specification can be aimed at a specific type of container.

❏ A deployment specification is a general mechanism to parameterize a deployment relationship.

❏ A specialized artifact.

```
+--------------------------+          +--------------------------+
|   <<deployment spec>>    |          |   <<deployment spec>>    | - - - + «deploy»
|         Name             |          |         Name             |       |
+--------------------------+          +--------------------------+       ↓
|       attributes         |
+--------------------------+
```

```
                                    +----------------------------------+
                                    |              Node                |
                                    |   +---------------------------+  |
+--------------------------+        |   |      <<artifact>>       📄 |  |
|   <<deployment spec>>    | - - -> |   |    DeployedArtifact       |  |
|         Name             |        |   +---------------------------+  |
+--------------------------+        +----------------------------------+
```
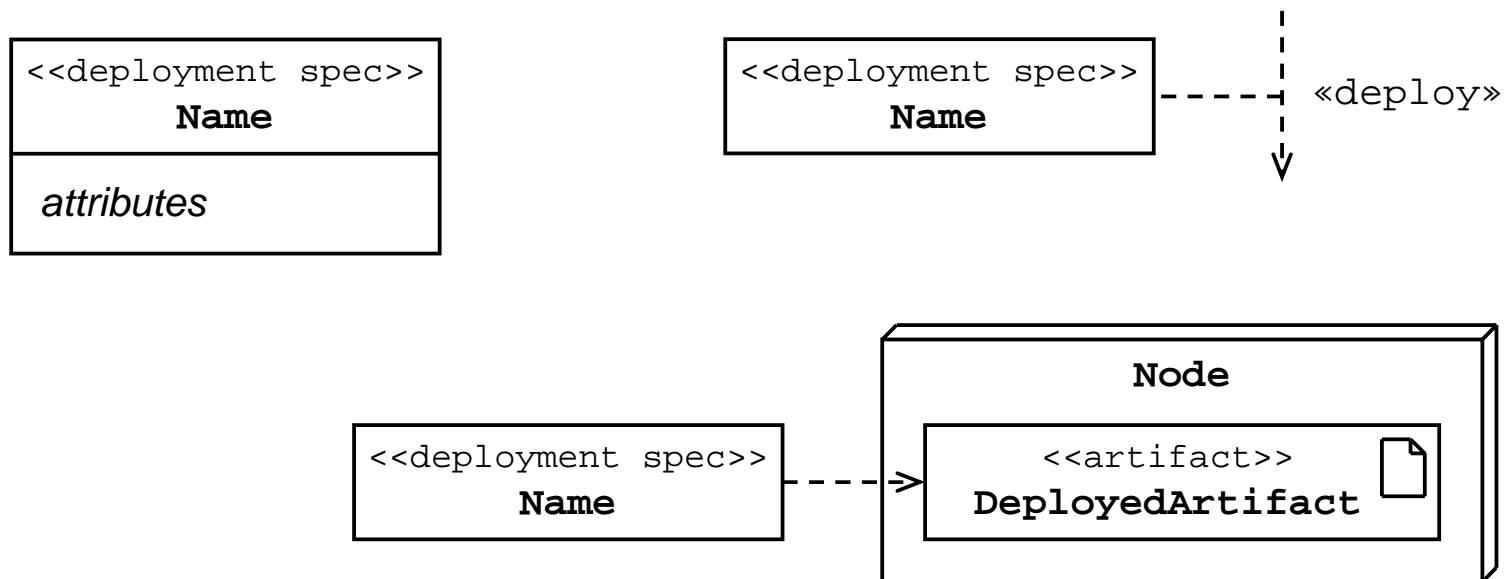
Examples of Deployment Specifications
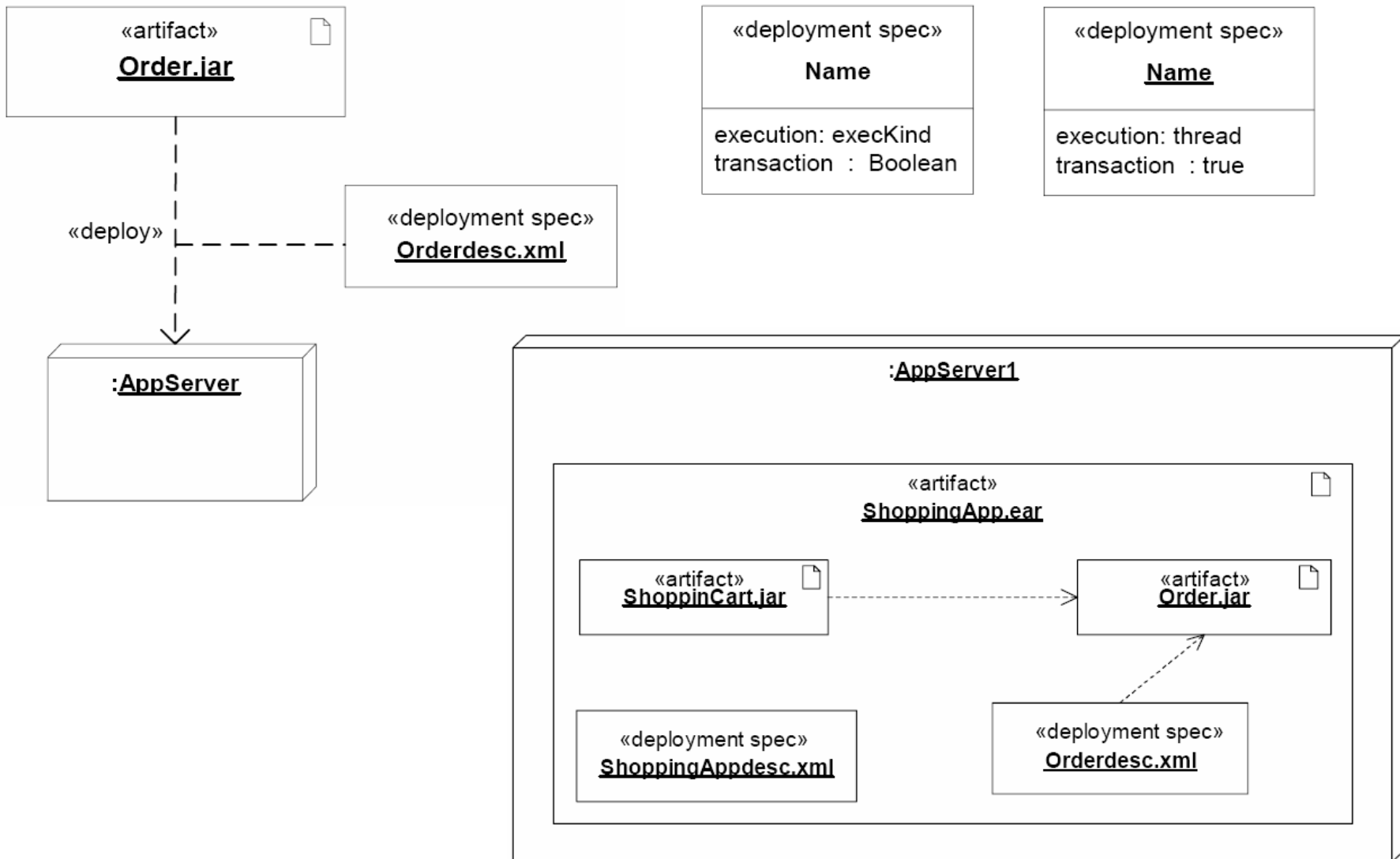
Object-Oriented Analysis and Modeling

Interactions

Interaction Model

→ **Defines the mutual interactions and collaboration of objects in certain situations.**

Diagram types:

- Sequence diagrams.

- Interaction overview diagrams.

- Communication diagrams.

- Timing diagrams.

- Interaction tables.

Used (mainly) in:

- Analysis and Design $\Rightarrow$ use case realization and interaction of objects.

Sequence Diagram

→ An interaction diagram which focuses on the message interchange between a number of lifelines.
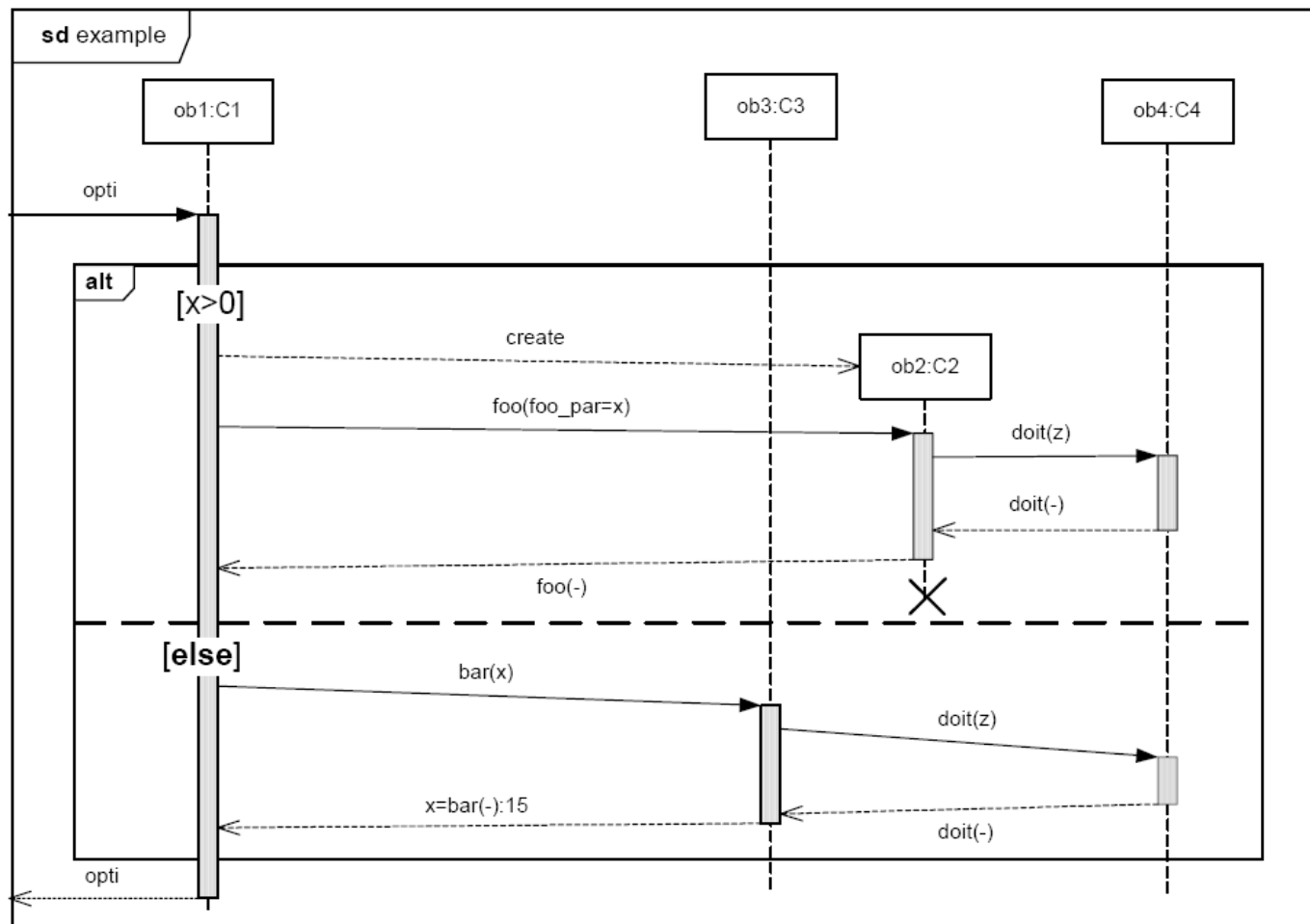
Interaction

→ A unit of behavior that focuses on the observable exchange of information between connectable elements.

❑ Focus on exchanging messages between the connectable elements (of the classifier owning the interaction).

❑ An interaction comprises:

- lifelines,
- messages,
- interaction fragments,
- formal gates, and
- actions.

❑ The semantics expressed by *valid* and *invalid traces*.

- A *trace* is a sequence of event occurrences (such as send operation/signal event, receive operation/signal event or destruction event) in a model.

❑ A specialization of interaction fragment and of behavior.

Example of Interaction
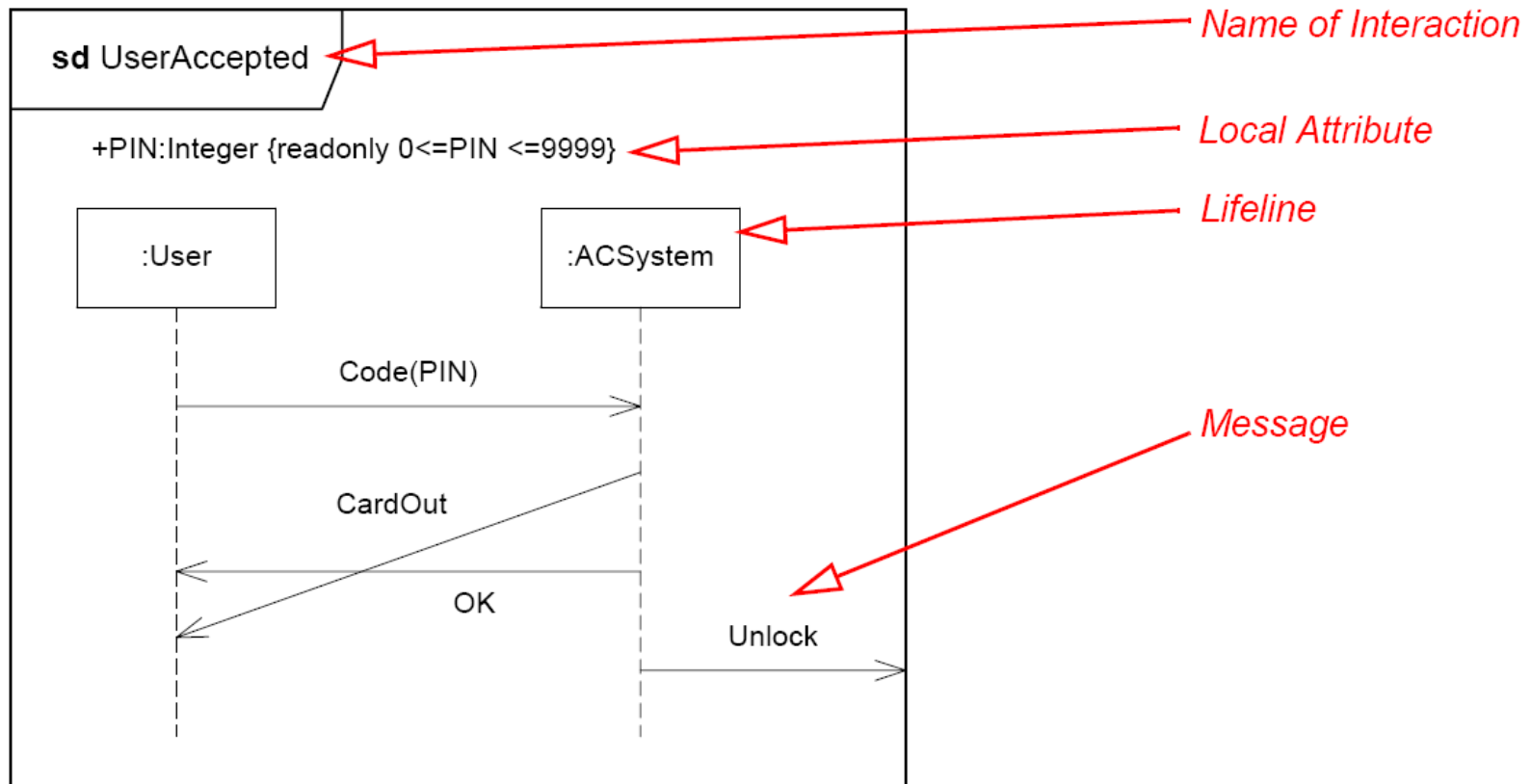
Frame

→ A rectangular frame around the diagram with a name in a compartment in the upper left corner.

❏ "sd" is used to determine the interaction diagram.

❏ Is used in all kinds of interaction diagrams.

```
┌─────────────────────────────┐
│ sd Diagram name  ╲          │
│────────────────── ╲         │
│                             │
│                             │
│                             │
│                             │
└─────────────────────────────┘
```
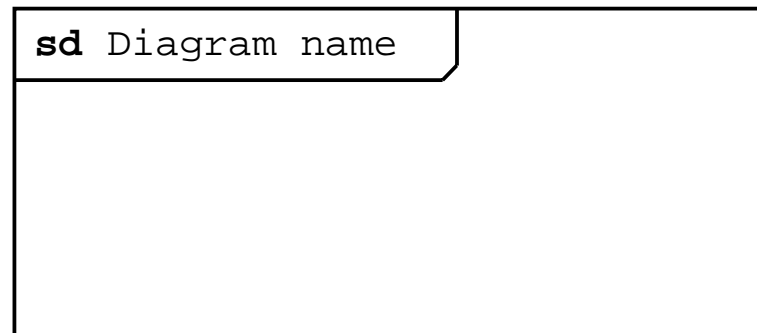
Lifeline

→   Represents an individual participant in the interaction.

❏   A reference to a connectable element.

❏   Lifelines represent only one interacting entity; have multiplicity 1.

   ●   If the referenced connectable element is multivalued, then the lifeline may have the **selector** that specifies which particular part is represented by this lifeline. If the selector is omitted, an arbitrary representative of the multivalued connectable element is chosen.

❏   Can refer to the interaction that represents the decomposition.

❏   Name format:

   lifelineident ::= ([*connectable-element-name* [‘[‘ *selector* ‘]’]]
      [: *class_name*] [*decomposition*]) | ‘self’

   selector ::= *expression*

   decomposition ::= ‘ref’ *interactionident* [’strict’]

```
Lifelineident
```
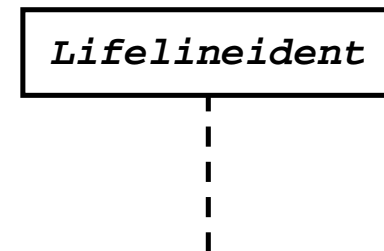
Part Decomposition

→   A description of the internal interaction of one lifeline relative to an interaction.

❏   A Lifeline has a class associated as the type of the connectable element that the lifeline represents. That class may have an internal structure and the part decomposition is an interaction that describes the behavior of that internal structure relative to the Interaction where the decomposition is referenced.

❏   The messages that go into (or go out from) the decomposed lifeline are interpreted as actual gates that are matched by corresponding formal gates on the decomposition.

Examples of Part Decompositions (1)

Examples of Part Decompositions (2)
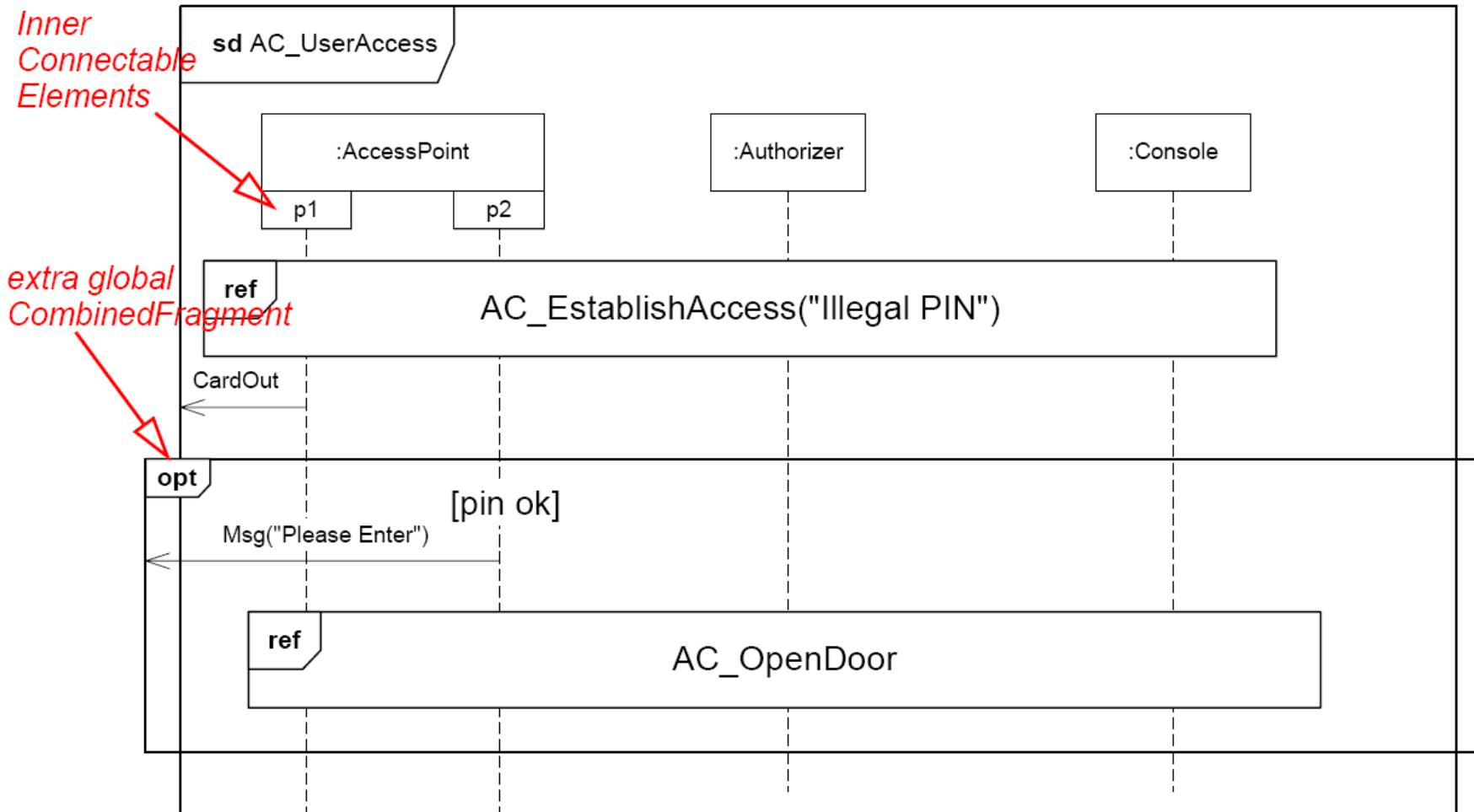
Message

→ Defines a particular communication between lifelines of an interaction.

❐ Specifies the kind of communication, sender and the receiver.

❐ Sorts of message:

- **Synchronous call** of an operation.
- **Asynchronous call** of an operation.
- **Asynchronous signal** – an asynchronous send action.
- **Create message** – the creation of another lifeline object.
- **Delete message** – the termination of another lifeline.
- **Reply** – a reply message to an operation call.
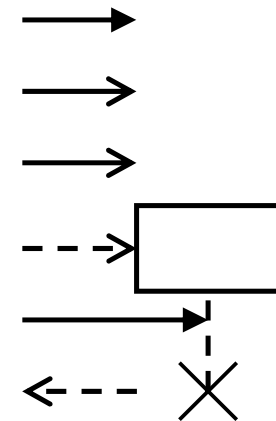
Message (cont.)

❑ Message connects *send event* and *receive event*.

❑ Kinds of a message:

- **Complete** – send event and receive event are present.

- **Lost** – send event present and receive event absent.

- **Found** – send event absent and receive event present.

- **Unknown** – send event and receive event absent (should not appear).

❑ Format of message name:

*messageident ::= ([attribute '='] signal-or-operation-name
['(' [argument [','argument]* ')'] [':' return-value]) | '*'*

*argument ::= ([parameter-name '='] argument-value) |
(attribute '=' out-parameter-name [':' argument-value]) | ' -'*

- '*' is a shorthand for more complex alternative combined fragment to represent a message of any type.

- '-' is used for undefined arguments.
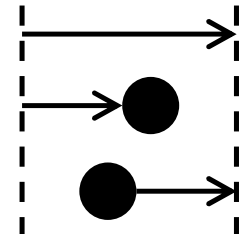
Examples of Message Names

mymessage(14, - , 3.14, "hello") // second argument is undefined

v=mymsg(16, variab):96 // a reply message assigning the return value 96 to v

mymsg(myint=16) // the input parameter 'myint' is given the value 16

Combined Fragment

→ Defines an "expression" of interaction fragments.

❑ A combined fragment is defined by an *interaction operator* and corresponding *interaction operands*.

❑ A compact and concise manner to define a number of traces.

❑ An *interaction operand* is an interaction fragment with an optional guard expression.

- Only the interaction operands with a guard evaluated to true at this point in the interaction will be considered for the production of the traces for the enclosing combined fragment.

- The order of the interaction operands is given by their vertical positions.

❑ *Alternatives (alt)*

- A choice of behavior. At most one of the operands will be chosen.

- The chosen operand must have an explicit or implicit guard expression that evaluates to true. (An implicit true guard is implied if the operand has no guard.)

- An operand guarded by *else* is applied if no other operand is chosen.

Combined Fragment (cont.)

❒ *Option (opt)*

- A choice of behavior where either the (sole) operand happens or not.

- Semantically equivalent to an alternative combined fragment where there is one operand.

❒ *Break (break)*

- The operand is a scenario that is performed instead of the remainder of the enclosing interaction fragment.

- When the guard of the break operand is false, the break operand is ignored and the rest of the enclosing interaction fragment is chosen.

- Should cover all lifelines of the enclosing interaction fragment.

❒ *Parallel (par)*

- A parallel merge between the behaviors of the operands.

- The occurrence specifications of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

- *Coregion* is used as a simplified form of the parallel fragment.

Combined Fragment (cont.)

□ *Weak sequencing (seq)*

- The ordering of occurrence specifications within each of the operands are maintained in the result.

- Occurrence specifications on different lifelines from different operands may come in any order.

- Occurrence specifications on the same lifeline from different operands are ordered such that an occurrence specification of the first operand comes before that of the second operand.

□ *Strict sequencing (strict)*

- A strict ordering of the operands on the first level.

- Therefore occurrence specifications within contained combined fragments will not directly be compared with other occurrence specifications of the enclosing combined fragment.

□ *Negative (neg)*

- The combined fragment represents traces that are defined to be invalid.

Combined Fragment (cont.)

□ *Critical region (critical)*

- The traces of the region cannot be interleaved by other occurrence specifications (on those lifelines covered by the region).

- Therefore, the enclosed occurrence specifications must be continuous.

- Used mainly within parallel combined fragments.

□ *Ignore / Consider (ignore / consider)*

- Combined with other interaction operators.

- Ignore = there are some message types that are *not shown* within this combined fragment. These message types can be considered *insignificant* and are *implicitly ignored* if they appear in a corresponding execution.

- Consider = designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be ignored.

- Format: ('ignore' | 'consider') '{' *message-name* [',' *message-name*]* '}'

Combined Fragment (cont.)

□ *Assertion (assert)*

- The sequences of the operand of the assertion are the only valid continuations.

- All other continuations result in an invalid trace.

- Often combined with Ignore or Consider.

□ *Loop (loop)*

- The loop operand will be repeated a number of times.

- The guard may include a lower and an upper number of iterations of the loop as well as a boolean expression.

- Format: 'loop' ['(' *min* [',' *max*] ')']

```
 _____
| operator |                    |
|----------·                    |
| - - - - - - - - - - - - - - - -|
|                                |
| - - - - - - - - - - - - - - - -|
|                                |
|_____|
```
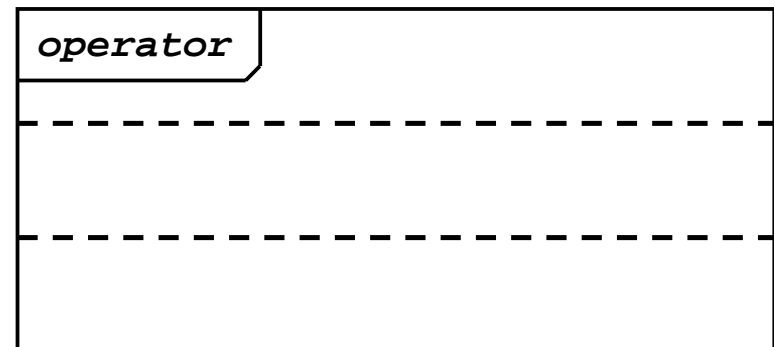
Examples of Combined Fragments (2)
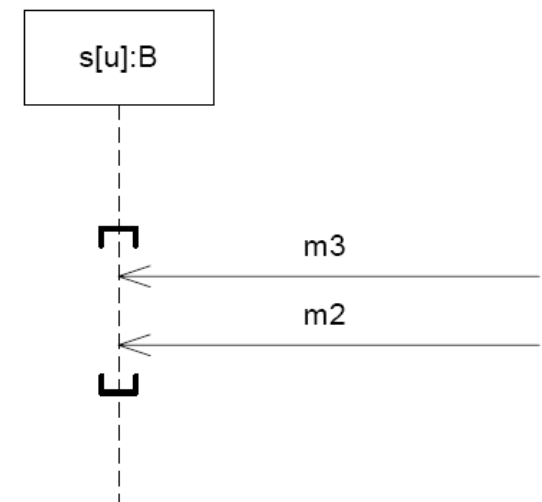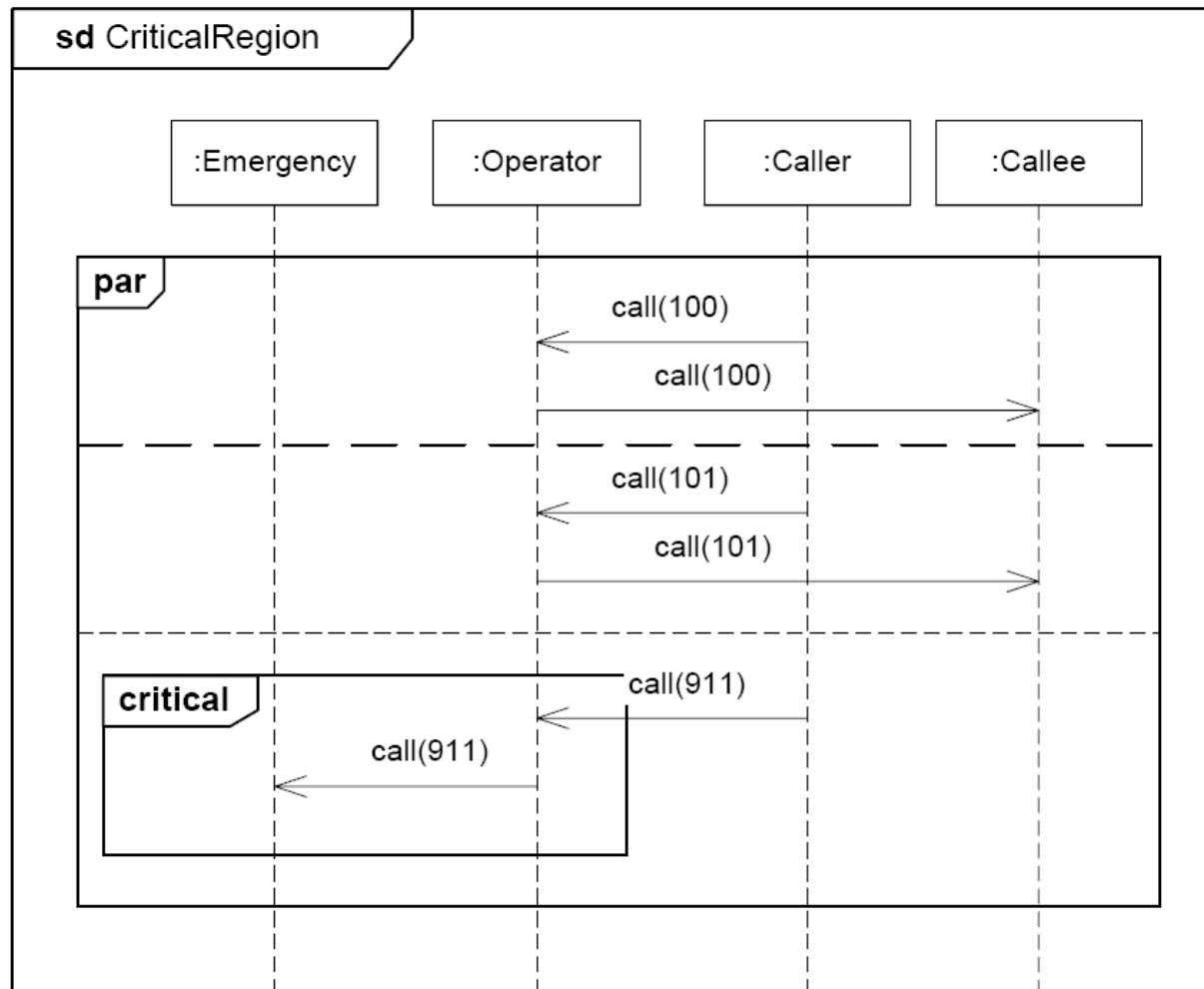
Interaction Use

→ A reference to an interaction.

❑ The interaction use is a shorthand for copying the contents of the referred interaction at the place where the interaction use occurs.

❑ The copying must take into account substituting parameters with arguments and connect the formal gates with the actual ones.

❑ Sharing an interaction as a portion of several other interactions.

❑ Name format:

name ::=[attribute-name '='] [collaboration-use '.'] interaction-name
    ['(' io-argument [',' io-oargument] ')'] [':' return-value*

io-argument ::= in-argument | 'out' out-argument]

```
┌─────┐
│ ref │              Name
└─────┴─────────────────────────────
│
└──────────────────────────────────
```
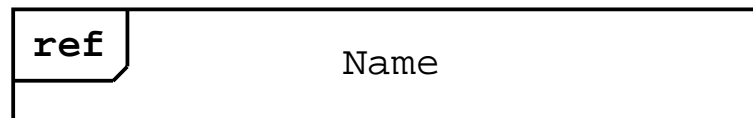
Examples of Interaction Uses (2)
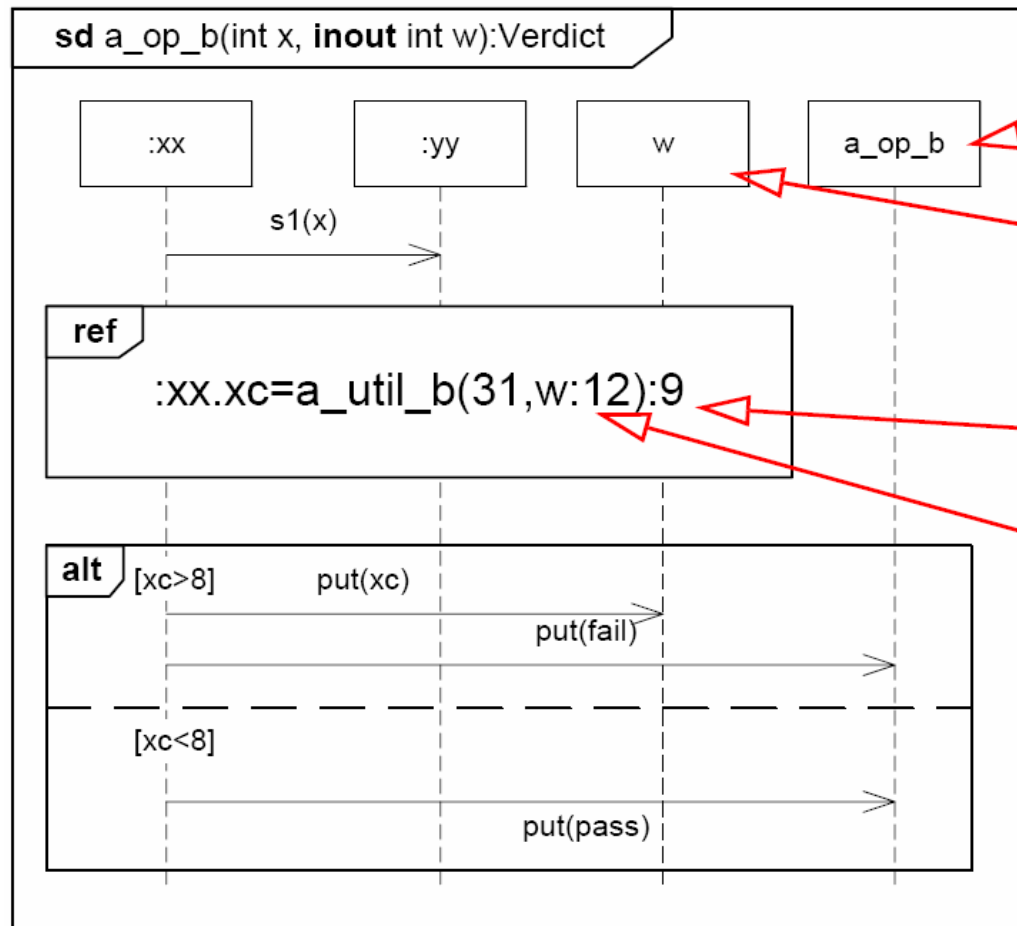
State Invariant

→ A runtime constraint on the participants of the interaction.

❒ It may be used to specify a variety of different kinds of constraints, such as values of attributes or variables, internal or external states, and so on.

❒ If the constraint is true, the trace is a valid trace; if the constraint is false, the trace is an invalid trace.

```
┌────────────┐
│            │
└────────────┘
      ┊
 { constraint }
      ┊
  ╭──────────╮
  │ State of │
  │the lifeline│
  ╰──────────╯
      ┊
```
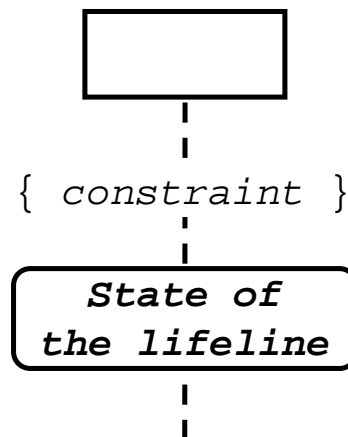
Example of State Invariant

Execution Specification
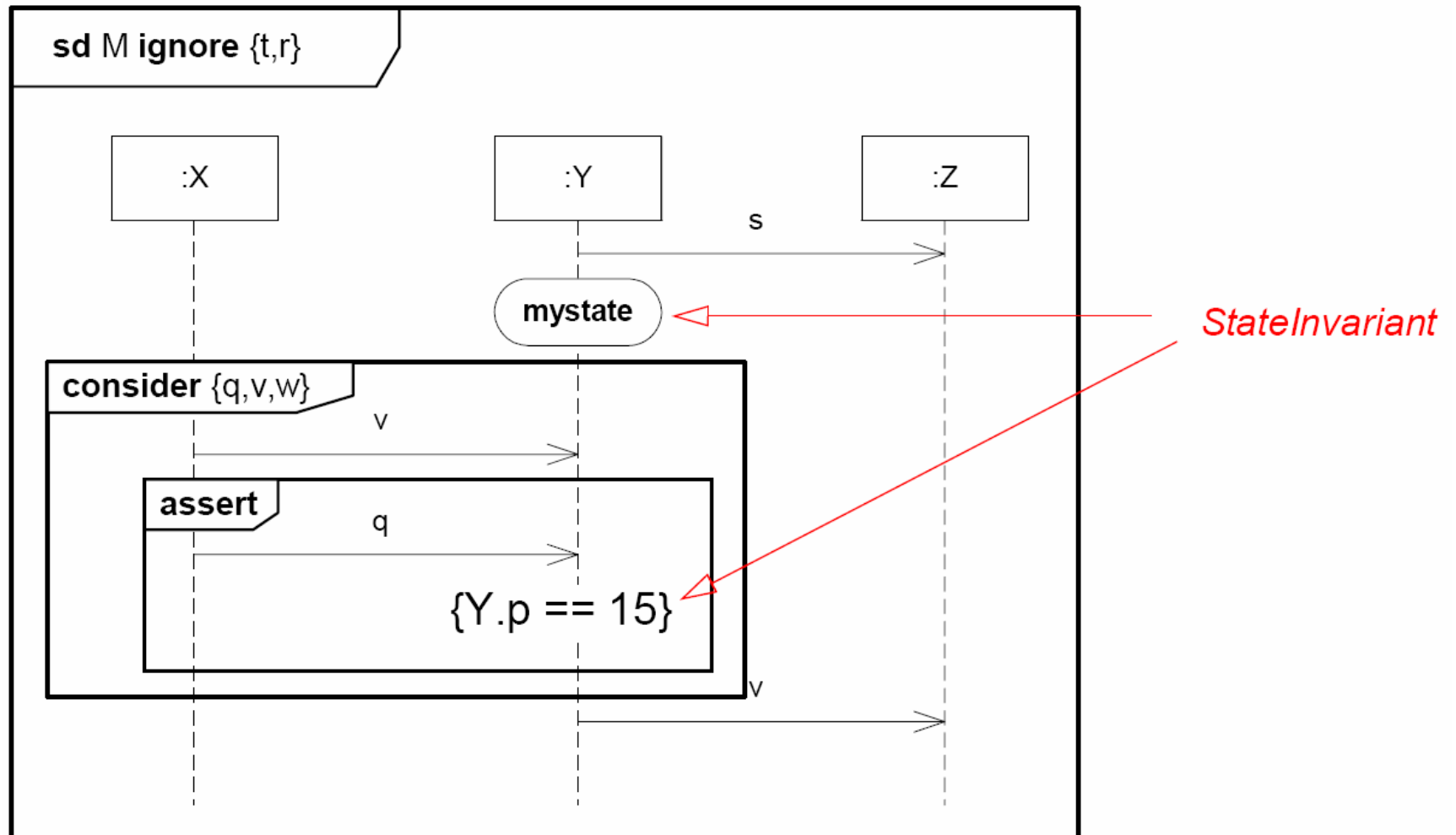
→ A specification of the execution of a unit of behavior or action within the lifeline.

❐ The duration of an execution specification is represented by the start execution occurrence specification and the finish execution occurrence specification.

- An *execution occurrence specification* represents a moment in time at which actions or behaviors start or finish.

Executed action
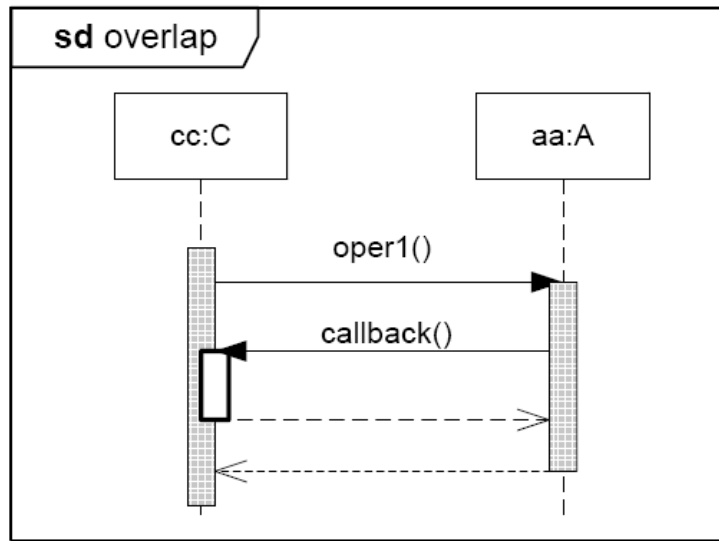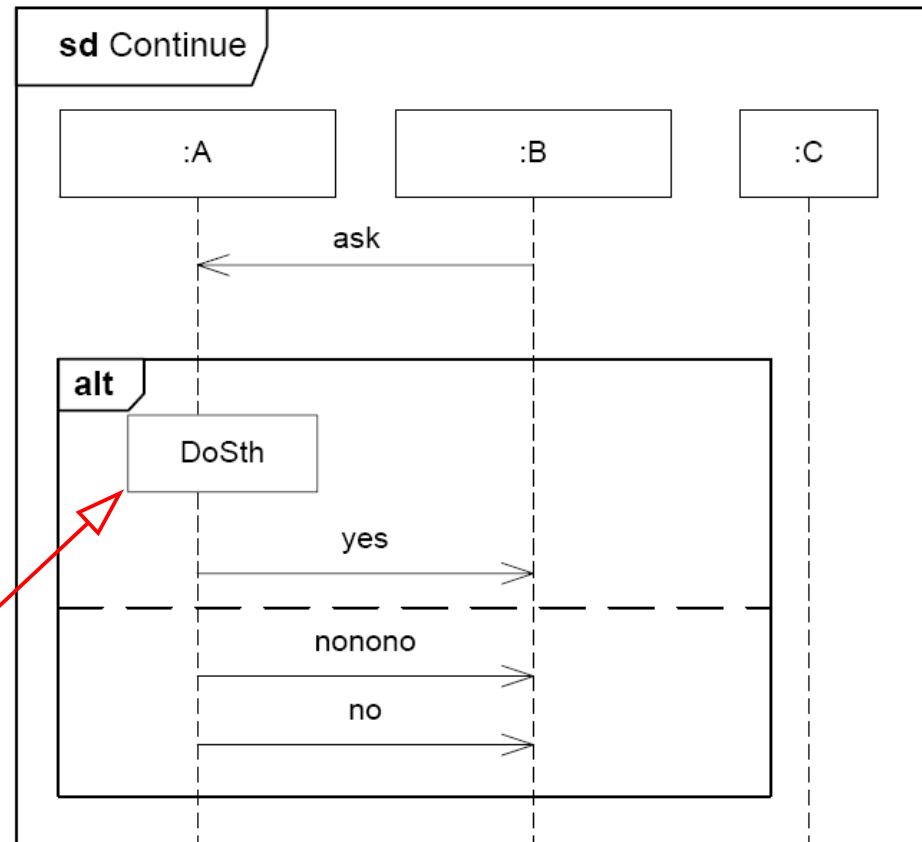
Continuations

→ A syntactic way to define continuations of different branches of an alternative or weak sequencing combined fragment.

❒ Continuation is intuitively similar to labels representing intermediate points in a flow of control.

❒ If an interaction operand of an alternative combined fragment ends in a continuation with name (say) *X*, only interaction fragments starting with the continuation *X* (or no continuation at all) can be appended.

Examples of Continuations
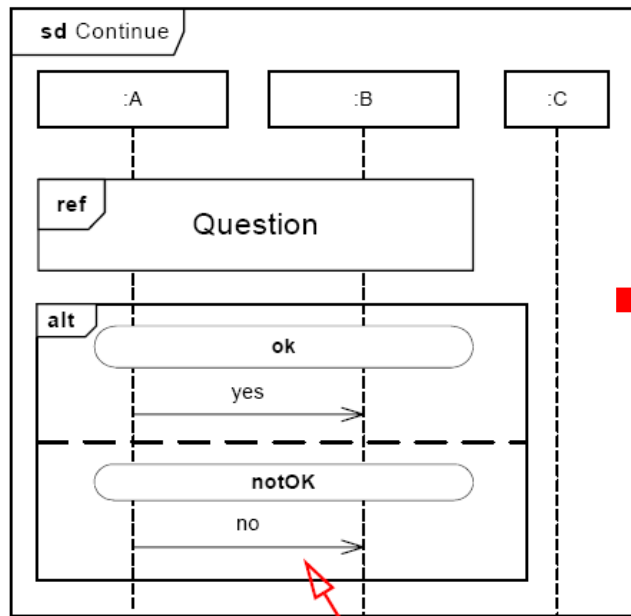
Continuation (setting==False)
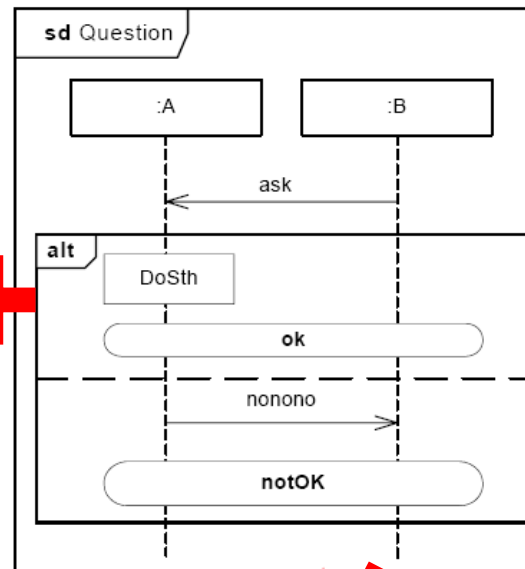
Continuation (setting==True)
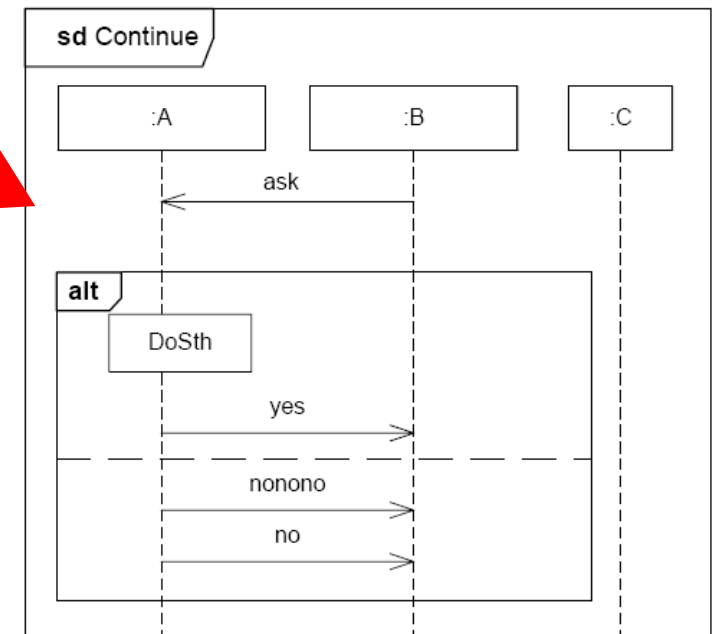
Destruction Event

→ Represents the destruction of the instance described by the lifeline.
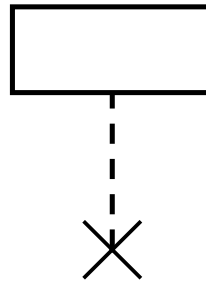
Gate

→ A connection point for relating a message outside an interaction fragment with a message inside the interaction fragment.

❑ Gates are connected through messages.

❑ Gates may be identified either by name (if specified), or by a constructed identifier formed by concatenating the direction of the message and the message name (e.g., "out_CardOut").

❑ Different roles:

- Formal gates on interactions.

- Actual gates on interaction uses.

- Expression gates on combined fragments.
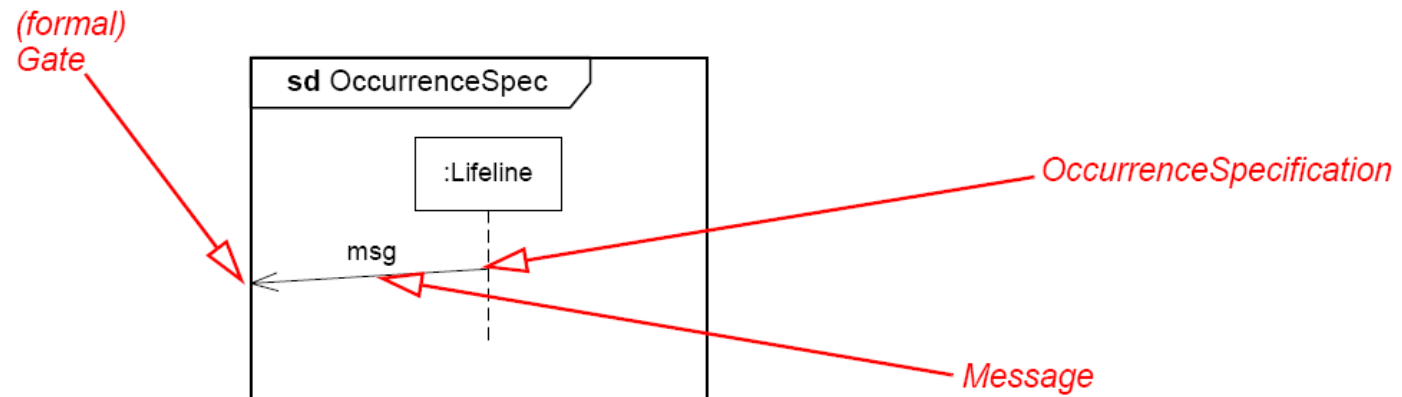
General Ordering

→ Represents a binary relation between two occurrence specifications, to describe that one occurrence specification must occur before the other in a valid trace.

❒ This mechanism provides the ability to define partial orders of occurrence specifications that may otherwise not have a specified order.

before ‣ ‣ ‣ ‣ ‣ ‣ ▶ ‣ ‣ ‣ ‣ ‣ ‣ *after*

Time Observation and Time Constraint

Time Observation

→ A reference to a time instant during an execution.

❒ It points out the element in the model to observe and whether the observation is when this model element is entered or when it is exited.

❒ They are usually named.

Time Constraint

→ Defines a constraint that refers to a time interval.

- A *time interval* defines the range between two time expressions; in interactions they usually refer to the time observations.

Duration, Observation and Duration Constraint

Duration

→ Defines a value specification that specifies the duration in time, i.e., temporal distance between two time instants—starting point in time and ending point in time.

Duration Observation

→ A reference to a duration during an execution.

Duration Constraint

→ Defines a constraint that refers to a duration interval.

- A *duration interval* defines the range between two durations.

Example of Timing Concepts
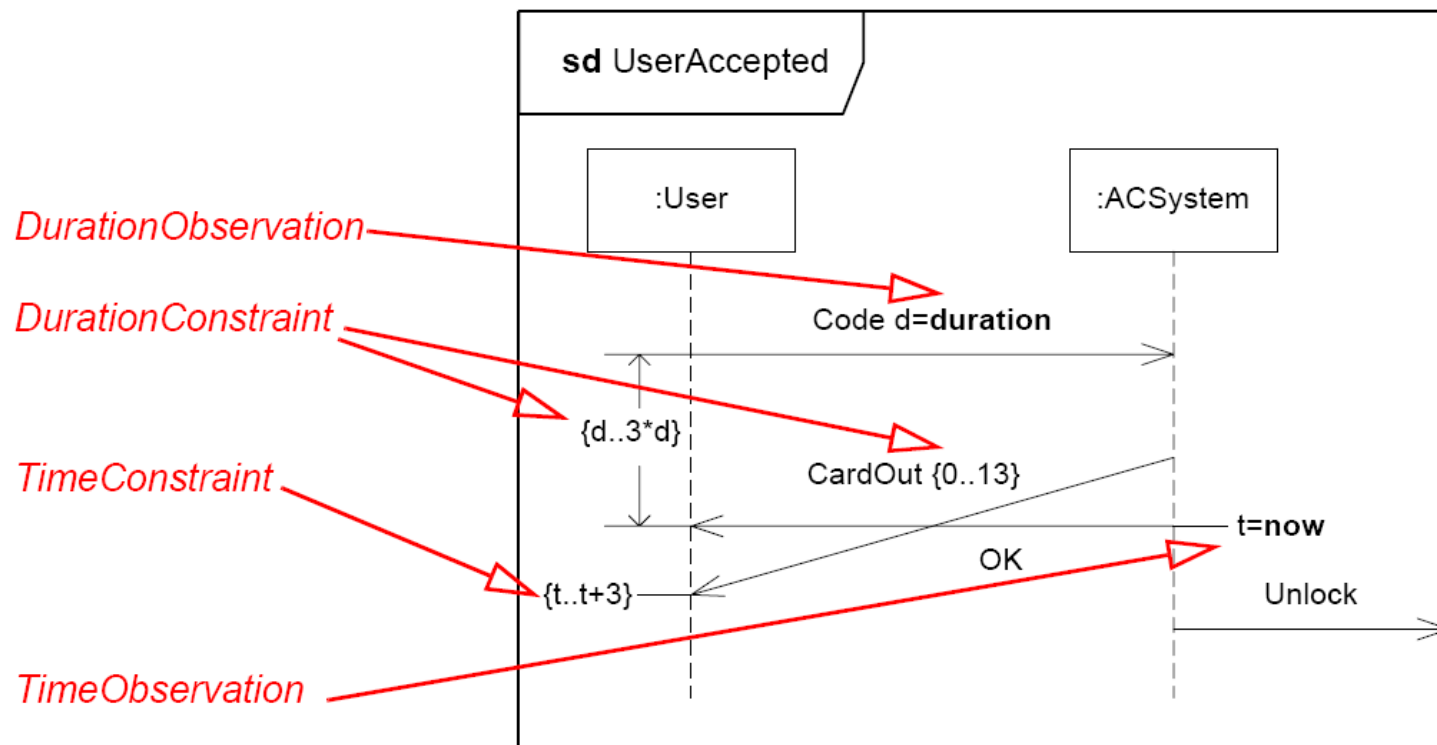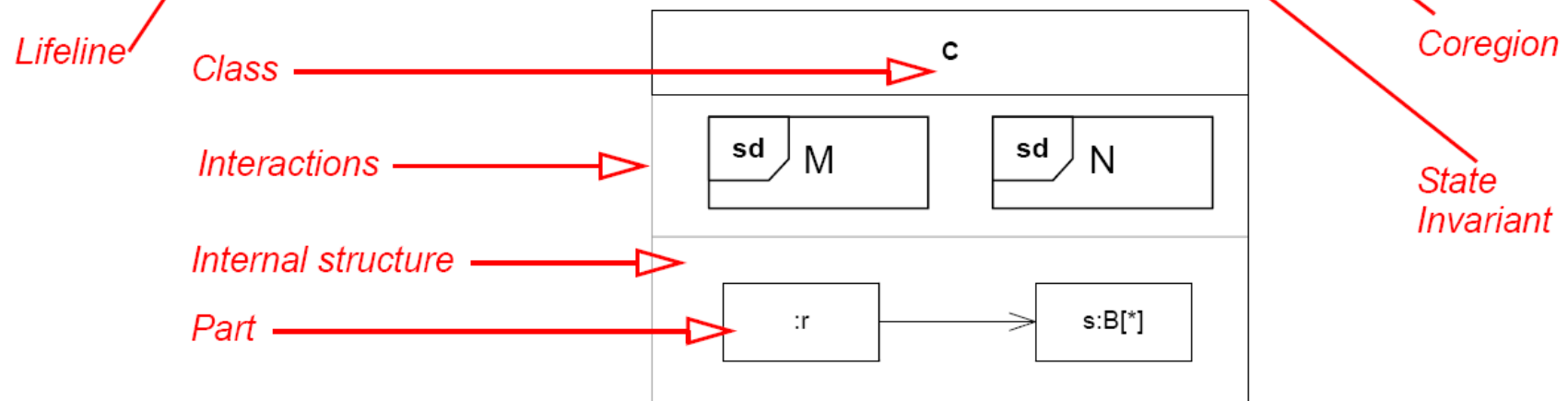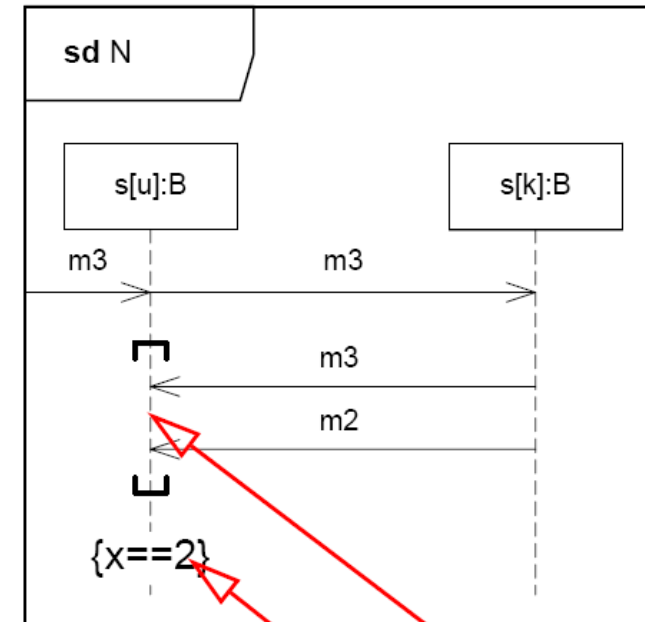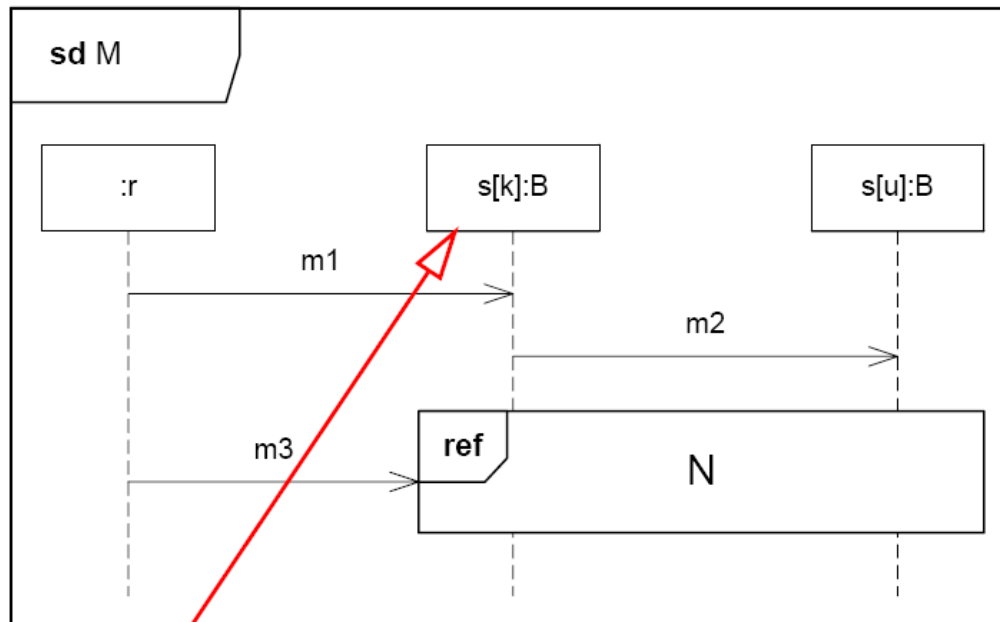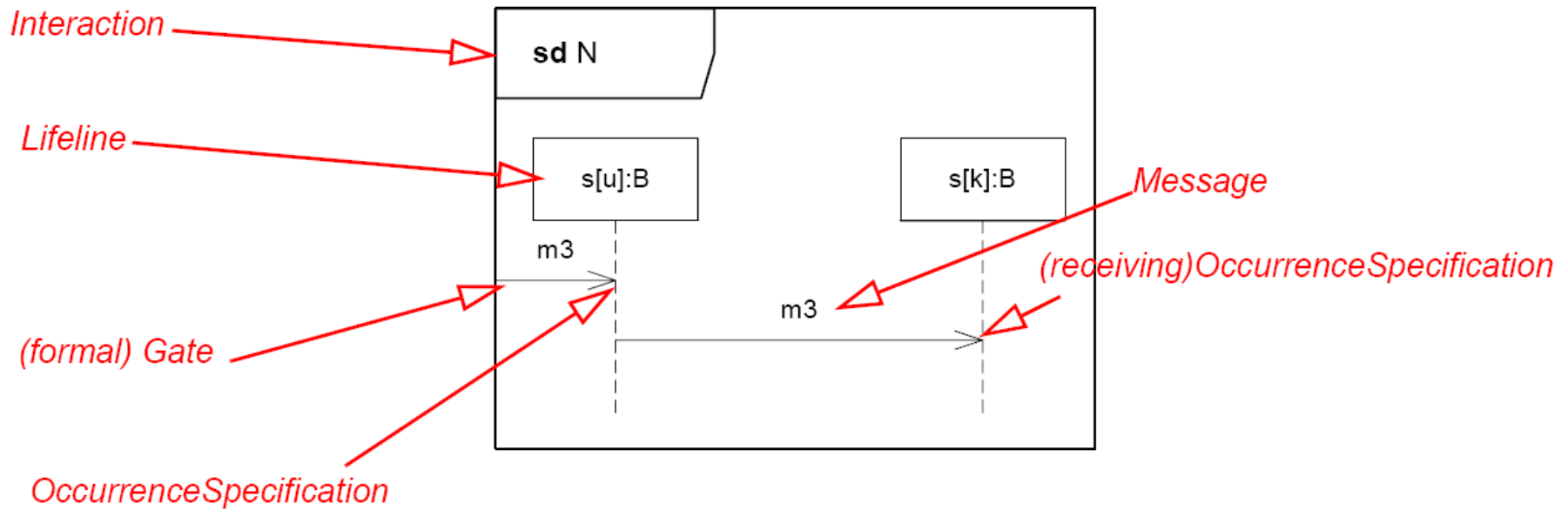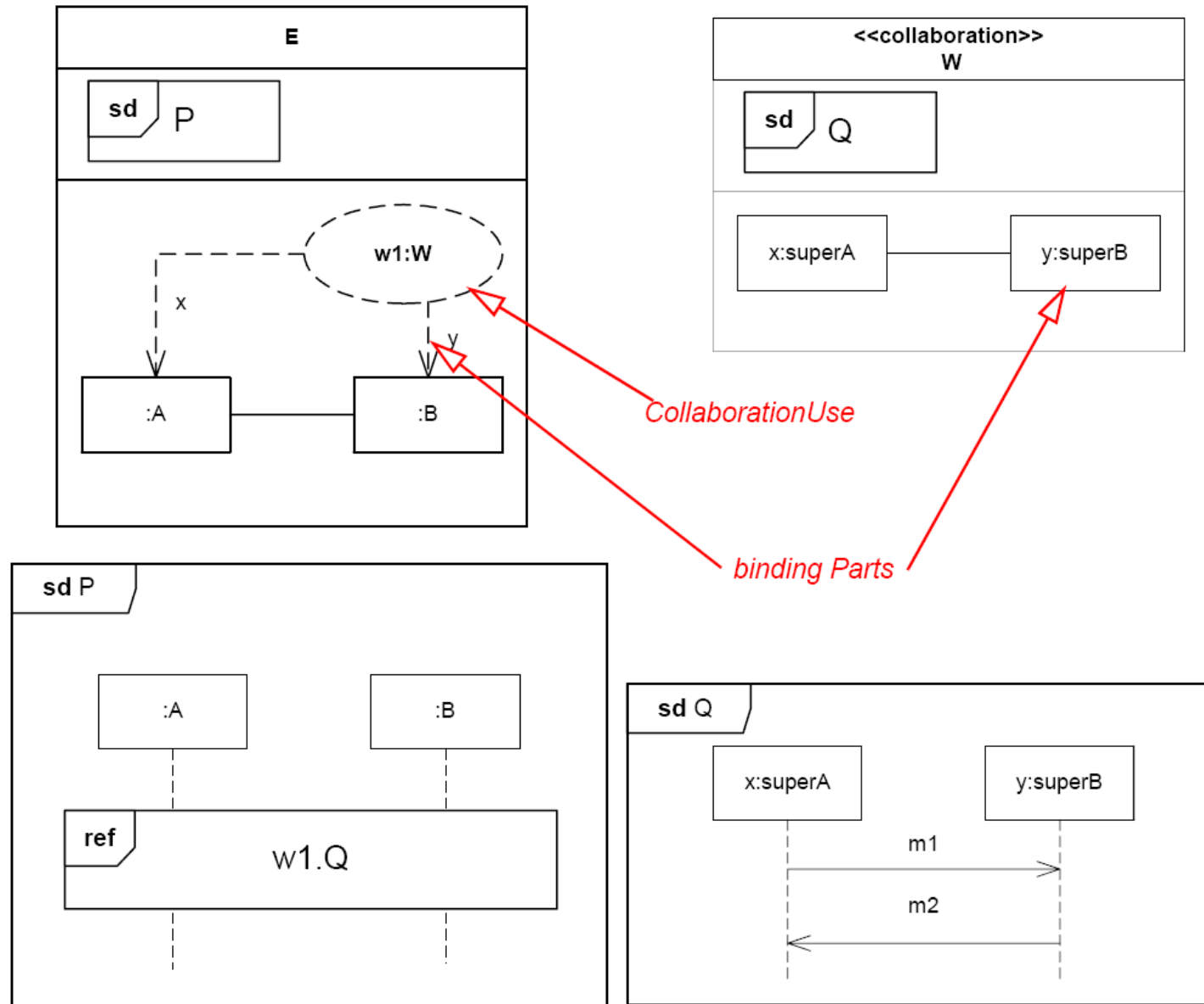
Communication Diagram

→ An interaction diagram which focuses on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central.

❑ Semantically corresponds to a simple sequence diagram.
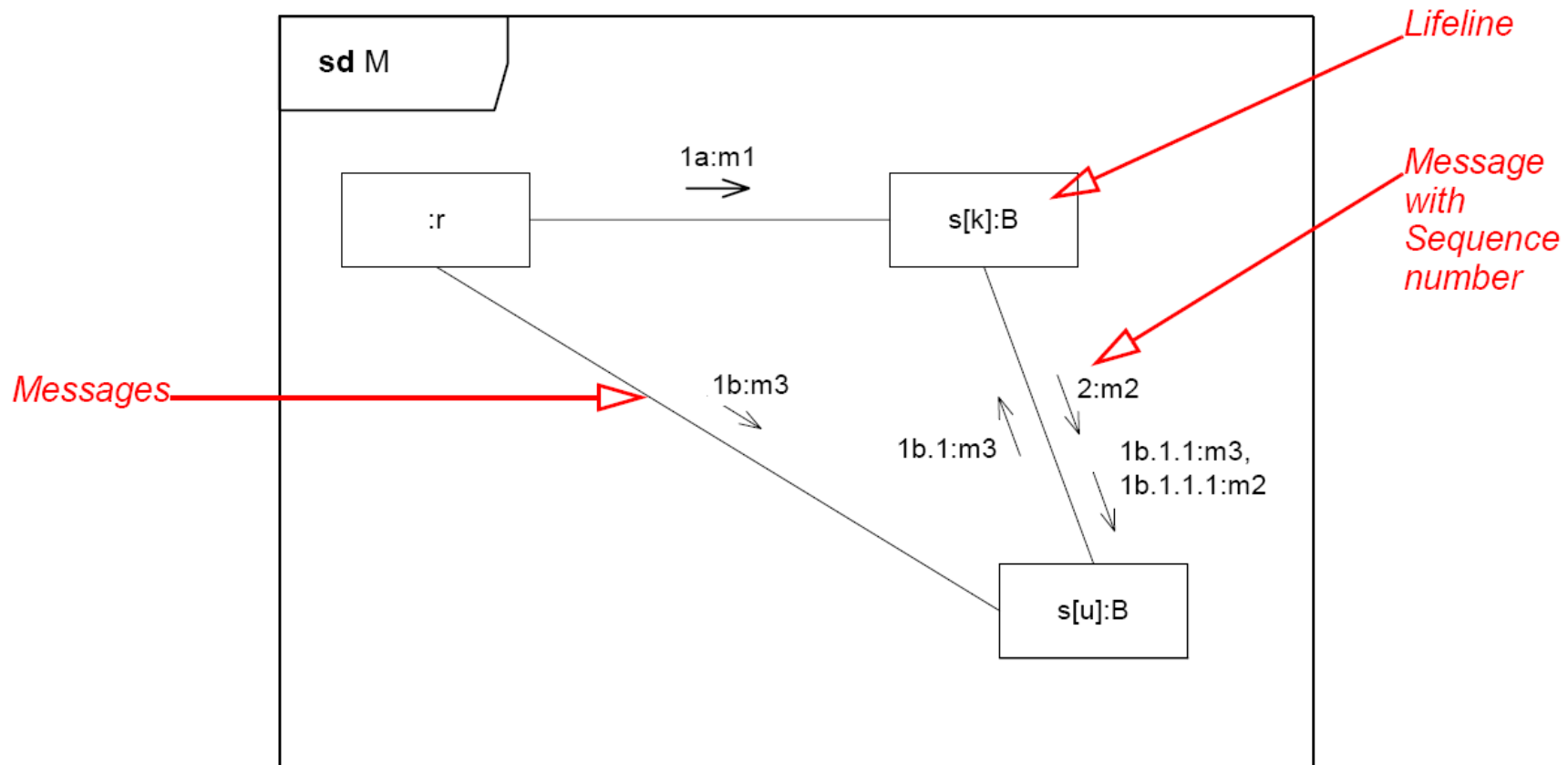
Lifeline

❑ Semantically identical to the lifeline from sequence diagrams.

❑ The format of the lifeline name ("lifelineident") identical to the lifeline from sequence diagrams.

❑ Communicating lifelines are linked by connectors.

```
                    ┌──────────────┐
                    │  Lifeline1   │
                    └──────────────┘
                     ╱            ╲
        ┌──────────────┐      ┌──────────────┐
        │  Lifeline2   │      │  Lifeline3   │
        └──────────────┘      └──────────────┘
```
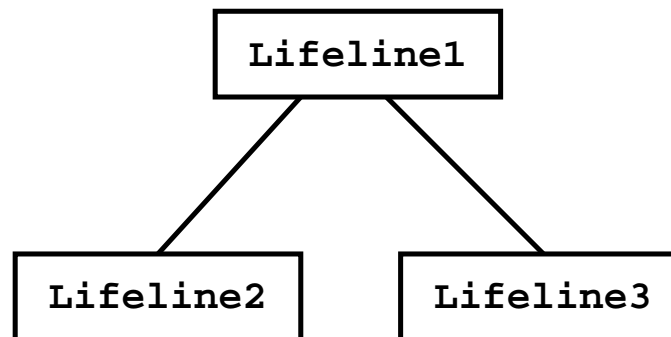
Message

- Semantically identical to the message from sequence diagrams.

- Arrow determines the communication direction.

- The message name is given by the following format:

  message-name ::= sequence-expression ':' messageident

  sequence-expression ::= sequence-term [sequence-term]*

  sequence-term ::= ['.' integer | name] [recurrence]

  recurrence ::= '*' ['||'] '[' iteration-clause ']' | '[' guard ']'

  - The *integer* represents the sequential order of the message within the next higher level of procedural calling.

    3.1.3 before 3.1.4 in 3.1

  - The *name* represents a concurrent thread of control.

    3.1a is concurrent with 3.1b within activation 3.1

  - The *recurrence* represents conditional or iterative execution.

    1.2*[i := 1..n]   3.5a[x > y]   4.3*||[1..3]

Interaction Overview Diagram

→ An interaction diagram which defines interactions through a variant of activity diagrams in a way that promotes overview of the control flow.

□ Interaction overview diagrams focus on the overview of the flow of control where the nodes are interactions or interaction uses.

□ The lifelines and the messages do not appear at this overview level.
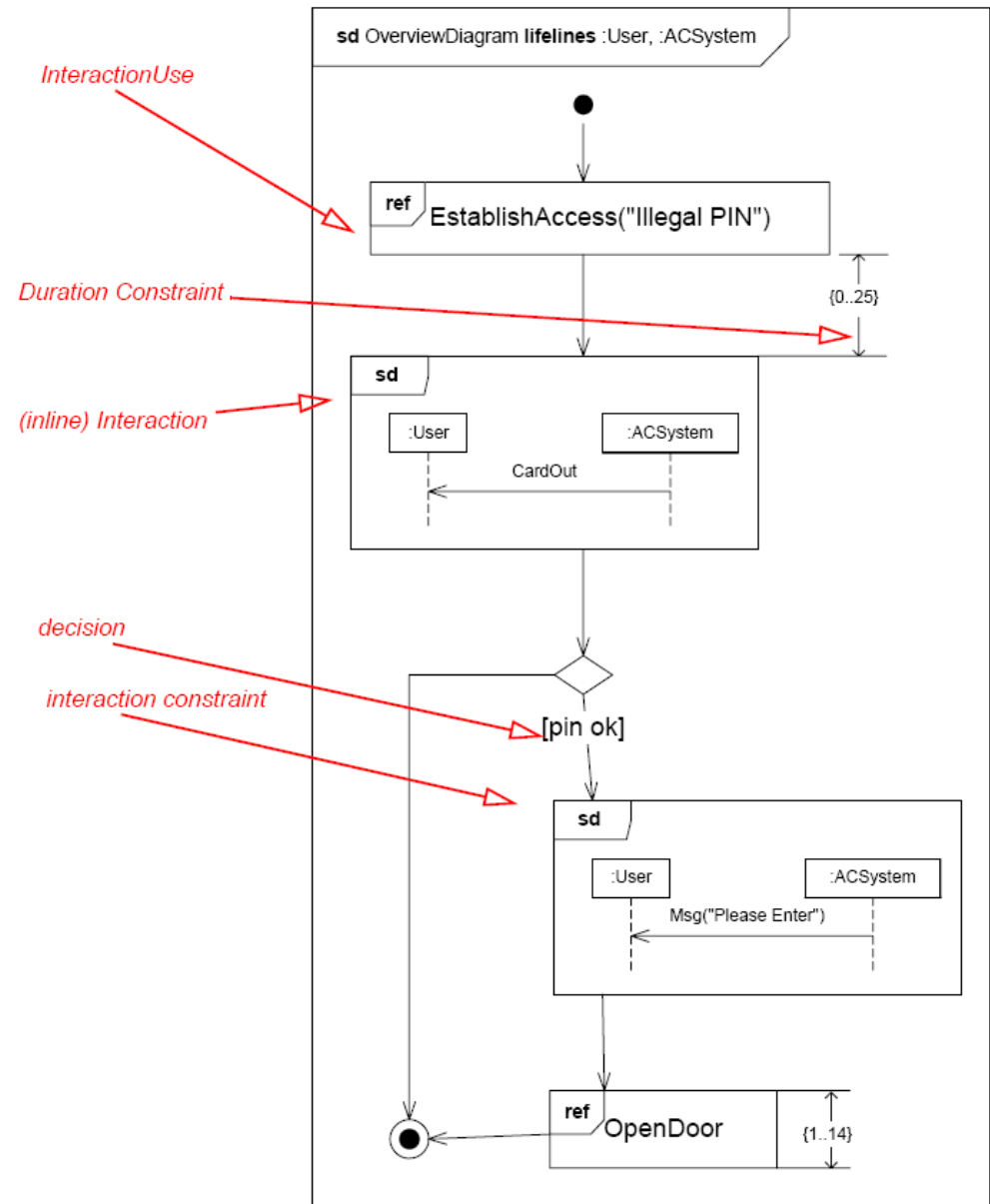
Differences from Activity Diagrams

1. In place of object nodes of activity diagrams, interaction overview diagrams can only have either (inline) interactions or interaction uses. Inline interaction diagrams and interaction uses are considered special forms of call behavior action.

2. Alternative combined fragments are represented by a decision node and a corresponding merge node.

3. Parallel combined fragments are represented by a fork node and a corresponding join node.

4. Loop combined fragments are represented by simple cycles.

5. Branching and joining of branches must in interaction overview diagrams be properly nested. This is more restrictive than in activity diagrams.

6. Interaction overview diagrams are framed by the same kind of frame that encloses other forms of interaction diagrams. The heading text may also include a list of the contained lifelines (that do not appear graphically).

Timing Diagram

→ An interaction diagram which is used to show interactions when a primary purpose of the diagram is to *reason about time*, focusing attention on time of occurrence of events causing changes in the modeled conditions of the lifelines.
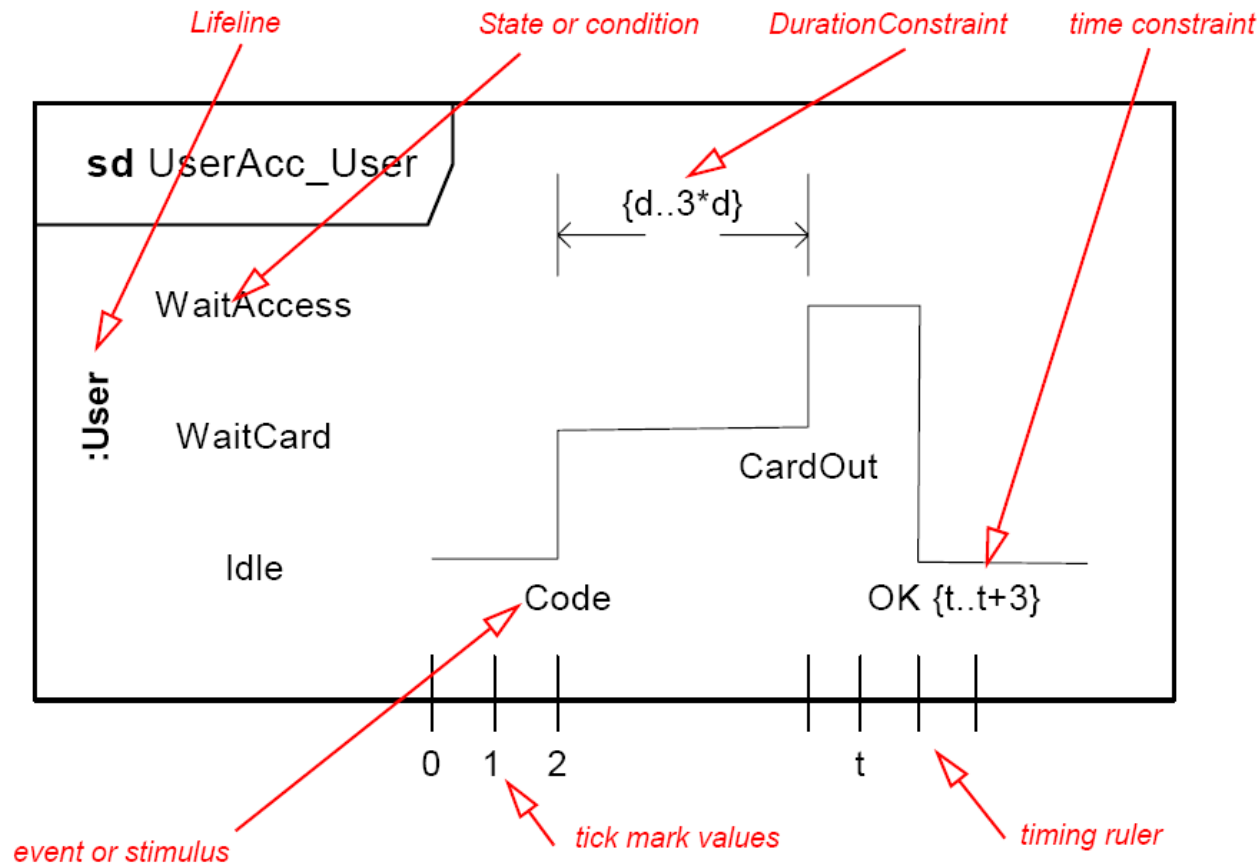
Lifeline

❒ Semantically identical to the lifeline from sequence diagrams.

❒ The format of the lifeline name ("lifelineident") is identical to the lifeline from sequence diagrams.

```
Lifeline1
```
_____

```
Lifeline2
```
_____

```
Lifeline3
```

Message

□ Semantically identical to the message from sequence diagrams.

□ The format of the message name ("messageident") is identical to the message from sequence diagrams.

□ The *message labels* can be used to denote that a message may be disrupted by introducing labels with the same name.

- Message labels are the notational shorthands used to prevent cluttering of the diagrams with a number of messages crisscrossing the diagram between lifelines that are far apart.
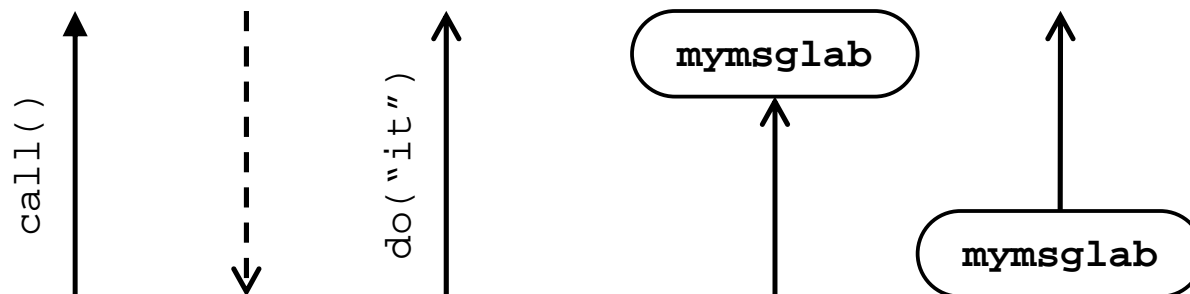
State or Condition Timeline

→ Representation of changing the state of the classifier or attribute, or some testable condition in time.

❏ It is also permissible to let the state-dimension be continuous as well as discrete. This is illustrative for scenarios where certain entities undergo continuous state changes, such as temperature or density.
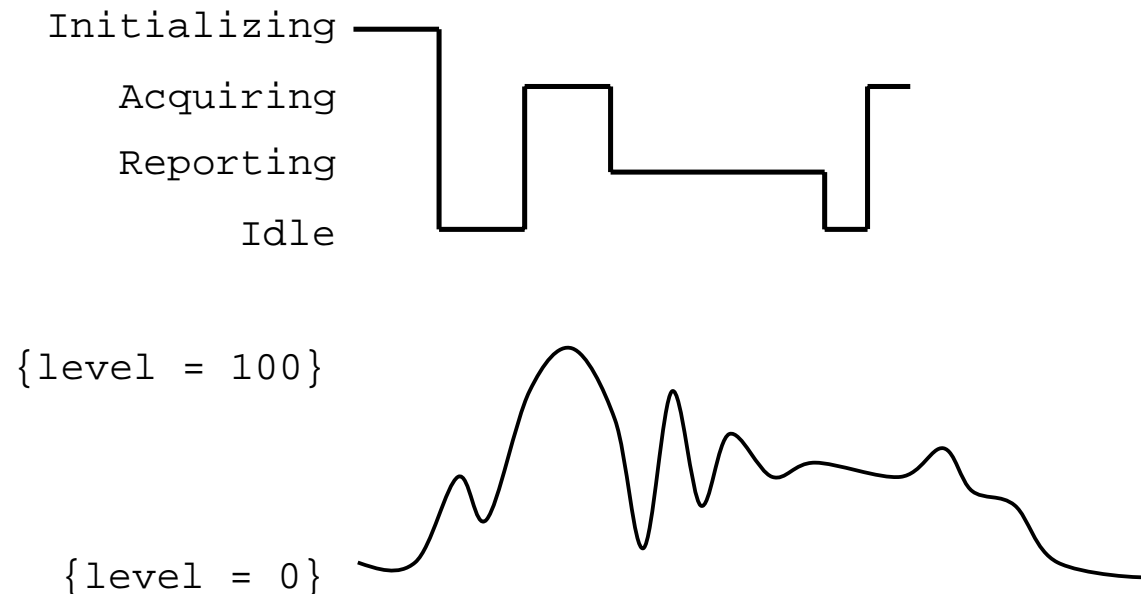
General Value Lifeline

→ Shows the value of the connectable element as a function of time.

❏ Value is explicitly denoted as text.

❏ Crossing reflects the event where the value changed.

```
x"FFFF"        x"00FF"
```

Additional Examples of Timing Diagrams (1)
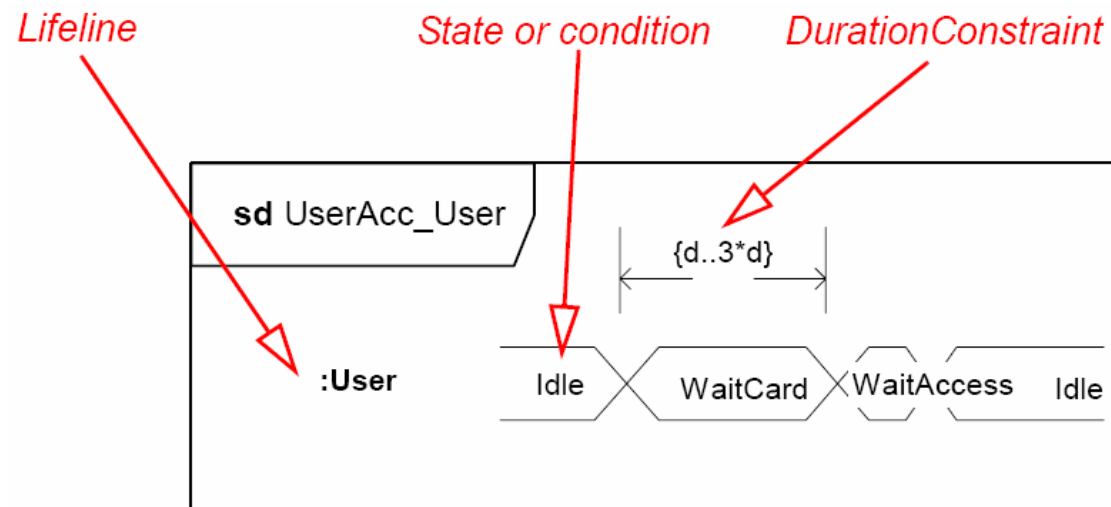
Additional Examples of Timing Diagrams (2)

Lifelines

State or condition

Duration Constraints

sd UserAccepted

{d..3*d}

WaitAccess

WaitCard

CardOut
{0..13}

Idle

Code

OK

Time Constraint

{t..t+3}

Message

NoCard

HasCard

Unlock

Duration Observation

d

t=now

Time Observation
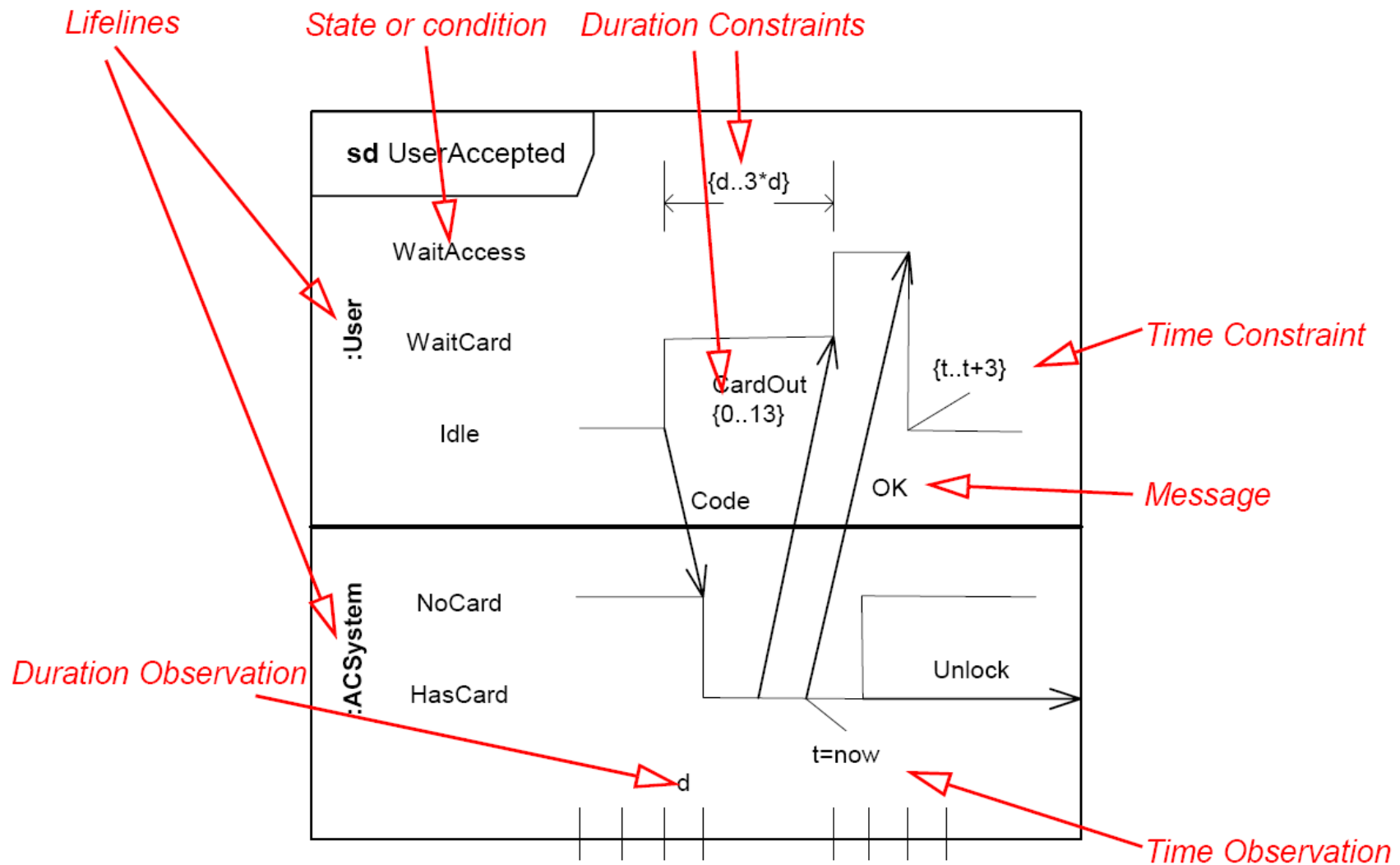
Object-Oriented Analysis and Modeling

Activities

Activity Model

→ **Specification of an algorithmic behavior.**

❏ Used to represent control flow and object flow models.

❏ *Executing activity* (of on object) is usually attached to a class or an operation.

❏ *Emergent activity* (of several objects) is usually attached to a package or a use-case.

Consists of:

- Activity diagrams.

- Element descriptions.

Used (mainly) in:

- Requirements $\Rightarrow$ algorithms of use cases.

- Analysis and design $\Rightarrow$ behavior of objects (their operations), subsystems, and components.

Example of Activity Diagram

Prepare Beverage — activity

initial

Find Beverage

[no coffee]

decision — [no cola]

[found coffee]

fork

Put Coffee in Filter

Add Water to Reservoir

Get Cups

Put Filter in Machine

join

[found cola]

control flow

Get Cans of Cola

action

Turn on Machine

Brew Coffee

merge

final

Pour Coffee

Drink
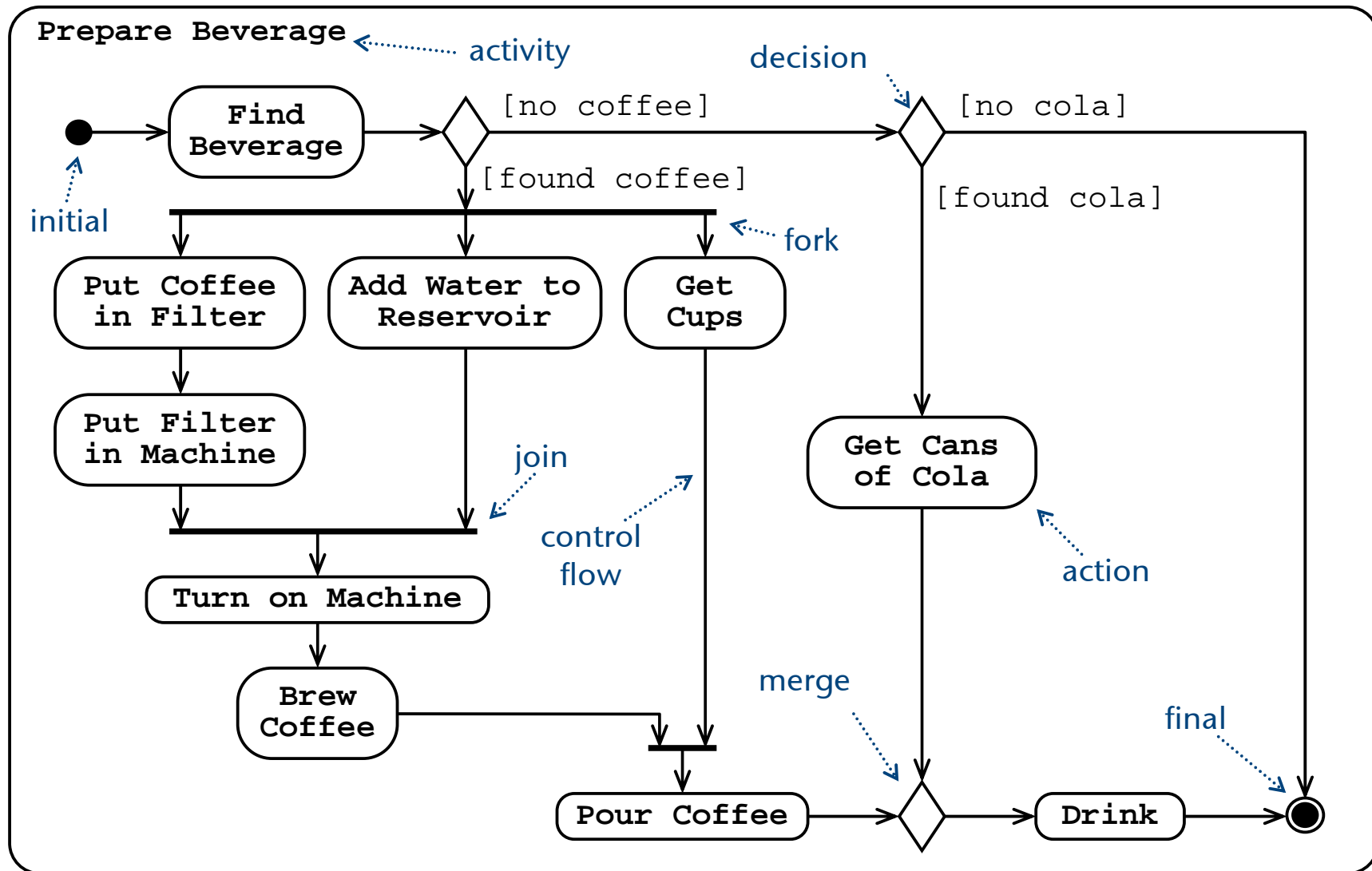
Activity

→ The specification of parameterized behavior as the coordinated sequencing of subordinate units whose individual elements are actions.

❏ Activities may describe procedural computation.

❏ The flow of execution is modeled as activity nodes connected by activity edges.

- A node can be the execution of a subordinate behavior.

- Activity nodes also include flow-of-control constructs, such as synchronization, decision, and concurrency control.

❏ Activities may be applied to organizational modeling for business process engineering and workflow.

❏ Formally, the semantics of activities is based on token flow.

- Tokens represent locus of control which execute the activity nodes and traverse along to the activity edges.

- There can be several distinct tokens in one execution of an activity.

- When a node completes execution, a token is removed from the node and tokens are offered to some or all of its output edges.

Activity (cont.)

❑ Activities can have inputs and outputs modeled by means of *activity parameters*.

- An input/output represents either as a single object or an object set.

❑ An activity has access to the attributes and operations of its context object and any objects linked to the context object transitively.

❑ An activity that is also a method of a behavioral feature has access to the parameters of the behavioral feature.
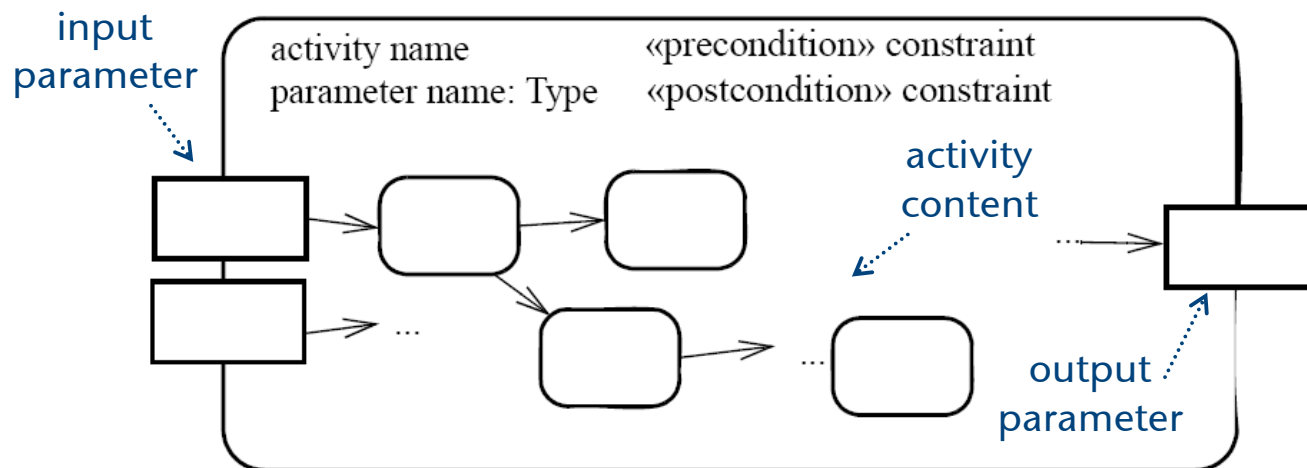
Process Order
Requested Order: Order

«precondition» Order complete
«postcondition» Order closed

«singleExecution»

[order rejected]

Requested Order

Receive Order

[order accepted]

Fill Order

Ship Order

Close Order

Send Invoice

Invoice

Make Payment

Accept Payment
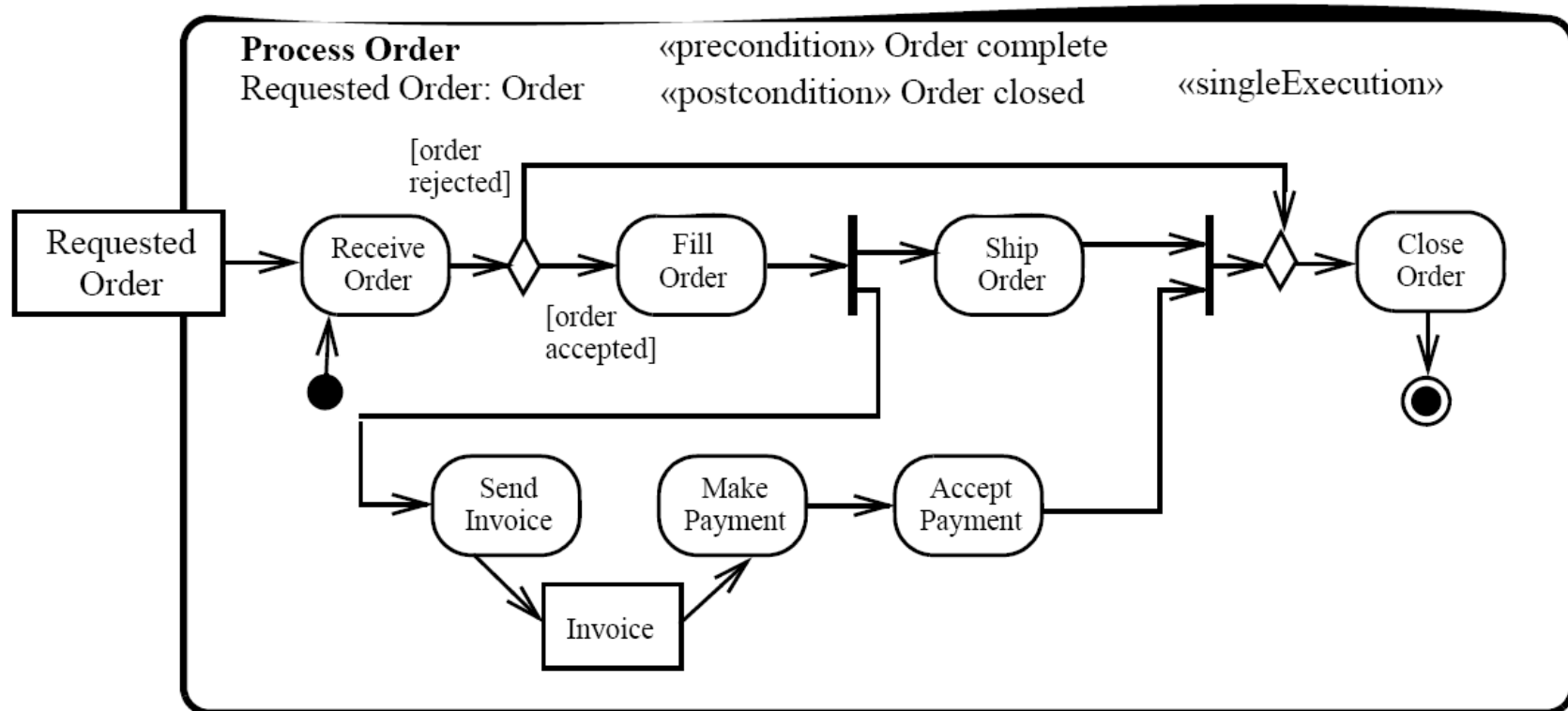
«activity»
Fill Order

costSoFar : USD
timeToComplete : Integer

suspend ()
resume ()

Examples of Activities (2)
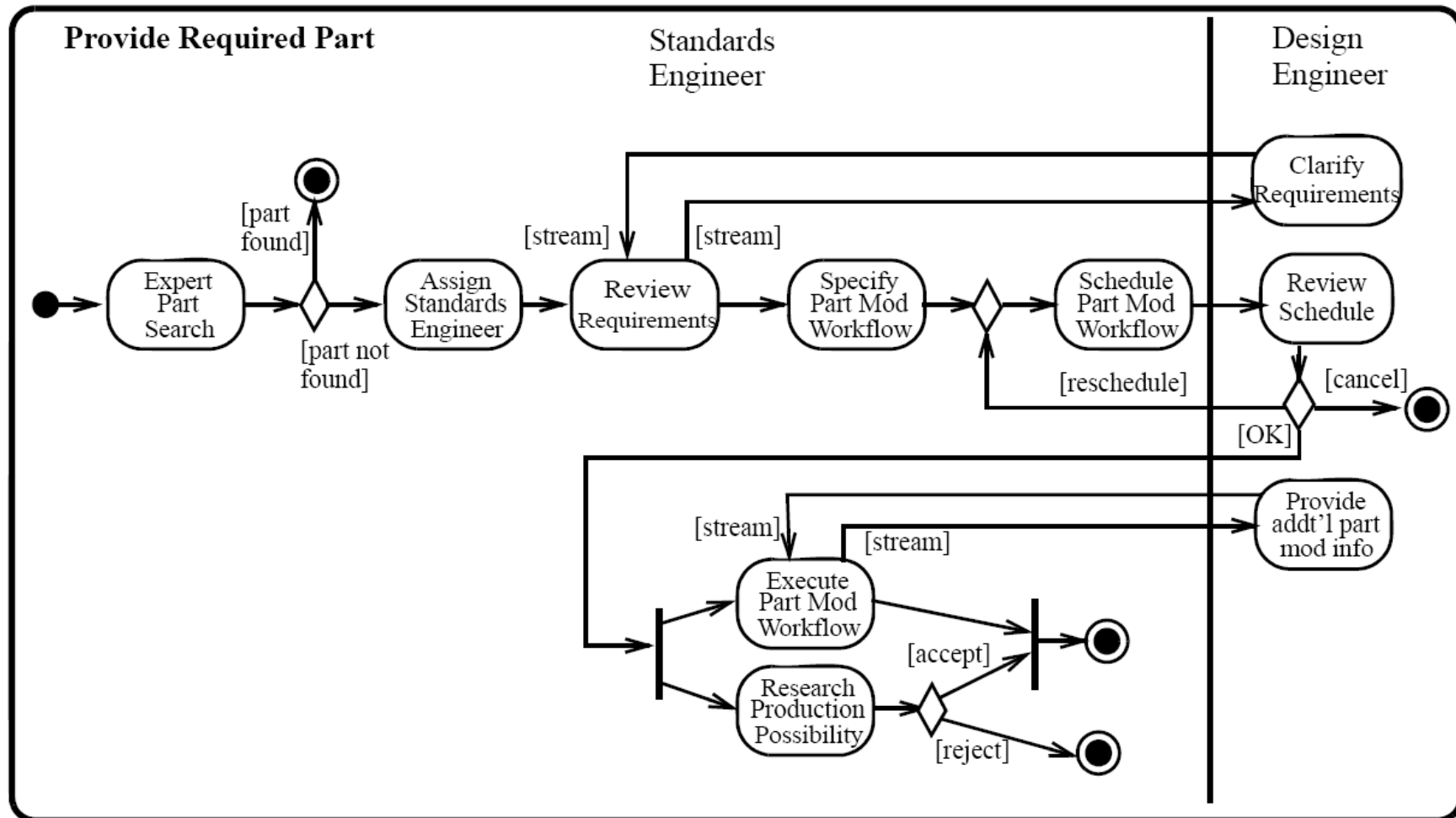
Action

→ The fundamental unit of executable functionality.

❑ The execution of an action represents some transformation or processing in the modeled system.

❑ Action can have inputs and outputs modeled by *pins*.

❑ Action can specify:

- *Local pre condition*—constraint that must be satisfied when execution is started.

- *Local post condition*—constraint that must be satisfied when execution is completed.

```
«localPrecondition»
constraint
```

```
Name
```

```
Name
```

```
« localPostcondition»
constraint
```
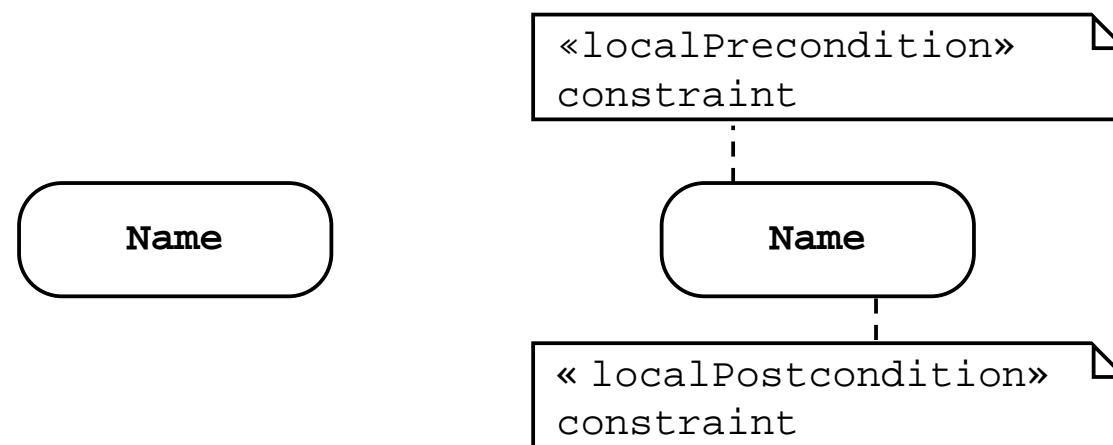
Some Special Kinds of Actions

- ❏ Call Behavior Action

  → Invokes a behavior directly.

  ```
  Behavior
  ```
  ```
  Activity
  ```

- ❏ Call Operation Action

  → Transmits an operation call request to the target object.

  ```
  operation
  ```
  ```
  Name
  (Class::operation)
  ```

- ❏ Accept Event Action

  → Waits for the occurrence of an event meeting the specified condition.
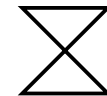
  ```
  Event
  ```
  ```
  Time Event
  ```

- ❏ Send Signal Action

  → creates a signal instance from its inputs, and transmits it to the target object

  ```
  Signal
  Type
  ```

- ❏ UML provides much more special action kinds for manipulating structural features, relations, communication, etc.

Control Flow

→ An activity edge that starts an activity node after the previous one is finished.

❐ An activity edge that only passes control tokens.

- Tokens offered by the source node are all offered to the target node.

❐ Can specify a guard condition which must evaluate to true for every token that is offered to pass along the edge.

❐ A connector (a small circle with the name of the edge in it) can also be used to simplify diagrams with many activity edges.

- Purely notational mechanism, it does not affect the model.

- Every connector with a given label must be paired with exactly one other with the same label on the same activity diagram.
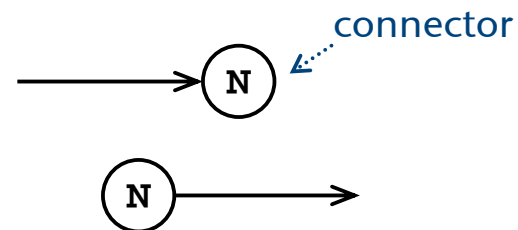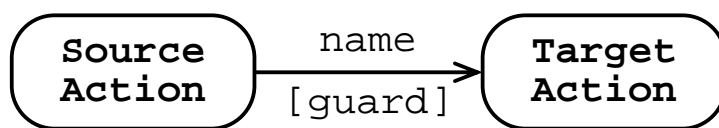
Examples of Control Flows
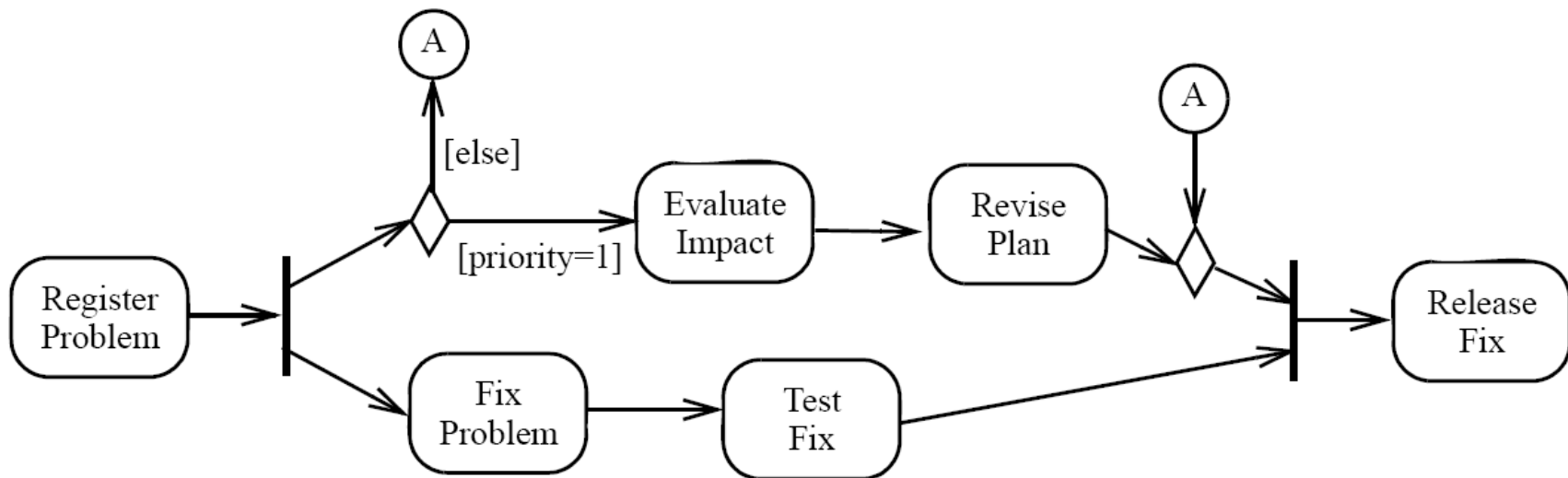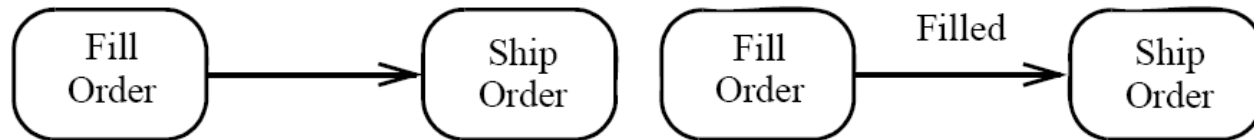
Initial Node

→ A control node at which flow starts when the activity is invoked.

❑ An initial node has no incoming edges.

❑ Only control edges can have initial nodes as source.

❑ A control token is placed at the initial node when the activity starts, but not in initial nodes in structured nodes contained by the activity.

- Tokens in an initial node are offered to all outgoing edges.

- If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node.
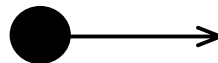
Activity Final Node

→ A final node that stops all flows in an activity.

❏ A token reaching an activity final node terminates the activity.

- It stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes.

- Terminating the execution of synchronous invocation actions also terminates whatever behaviors they are waiting on for return.

- Any behaviors invoked asynchronously by the activity are not affected.

❏ Has no outgoing edges.

❏ An activity may have more than one activity final node.

❏ If there is more than one final node in an activity, the first one reached terminates the activity.
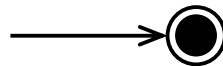
Flow Final Node

→ A final node that terminates a flow.

❑ Destroys all tokens that arrive at it.

❑ It has no effect on other flows in the activity.

❑ Has no outgoing edges.

⟶⊗
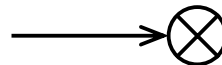
Examples of Final Nodes
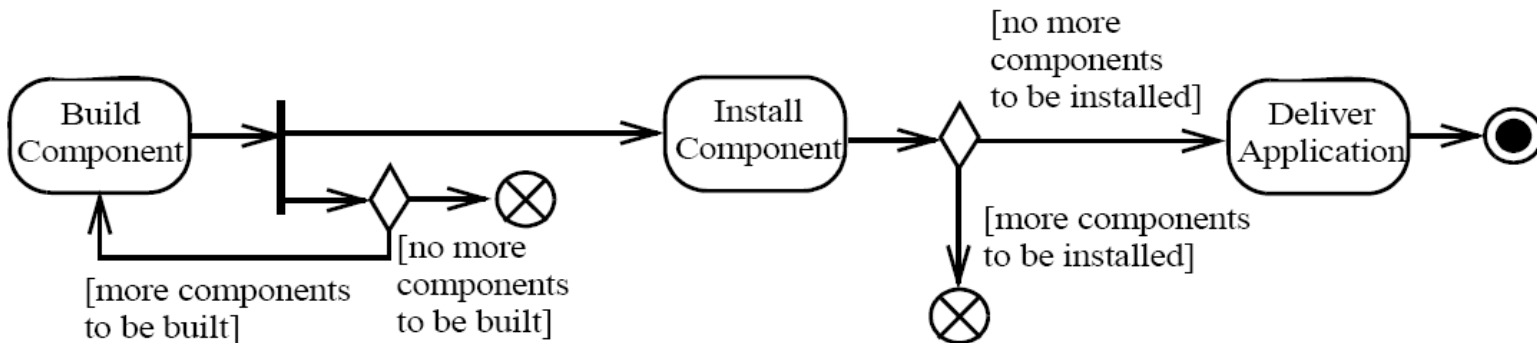
Object Node

→ An **abstract** activity node that indicates an instance of a particular *classifier*, possibly in a particular *state*, may be available at a particular point in the activity. Its concrete sub-elements are, e.g., Pin and Activity Parameter.

❏ All edges coming into or going out of object nodes must be object flow edges.

❏ Can specify multiplicity, ordering and selection criteria of the represented values (tokens).

❏ Multiple tokens containing the same value may reside in the object node at the same time. This includes data values.

  • A token in an object node can traverse only one of the outgoing edges.

object node with possible states and selection specification

object node with multiplicity and ordering

object node for tokens with signal as type

```
name
```

```
name:Classifier
[state, state, …]
```

```
name
```

```
name
```

```
«selection»
selection specification
```

```
{upperBound=2,
Ordering=FIFO}
```

Pins
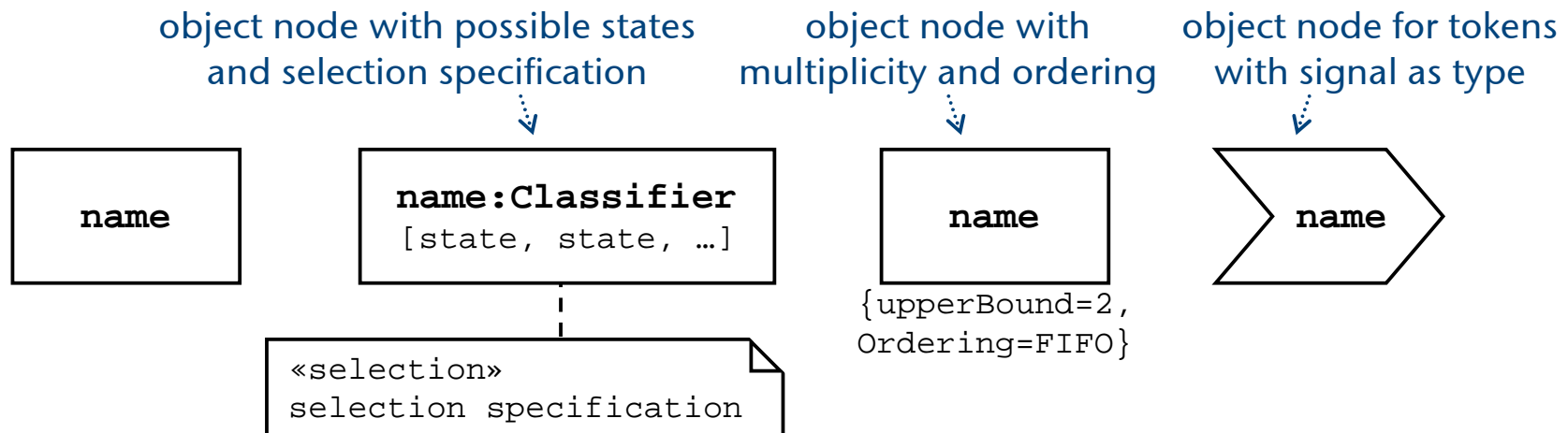
Input Pin

→ A specialized pin and object node that holds input values to be consumed by an action.

❒ Can specify name, type status and multiplicity.

- An action cannot start execution if an input pin has fewer values than the lower multiplicity.

- The upper multiplicity determines how many values are consumed by a single execution of the action.

```
p1       ↓  p2:T
              [state]
```
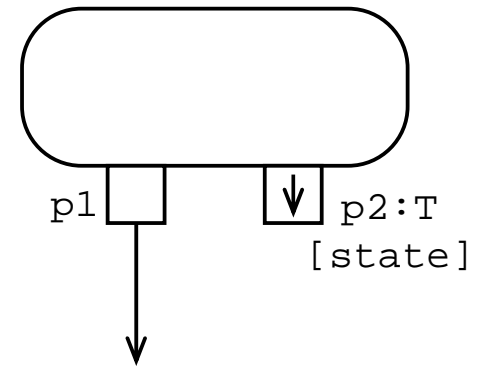
Output Pin

→ A specialized pin and object node that holds output values produced by an action.

❒ Can specify name, type and multiplicity.

- An action cannot terminate itself if an output pin has fewer values than the lower multiplicity.

- An action may not put more values in an output pin in a single execution than the upper multiplicity of the pin.

```
                  p2:T
p1        ↓  [state]
```
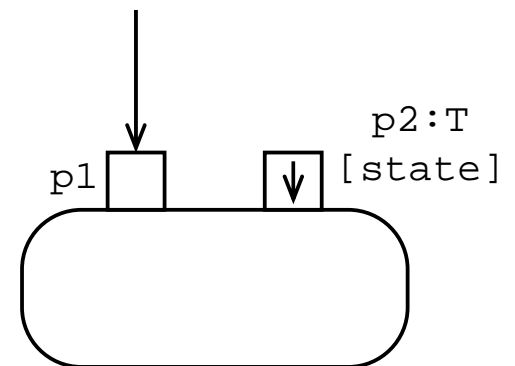
Pins (cont.)

Standalone pin notation representing
input and output pins together:

x

x
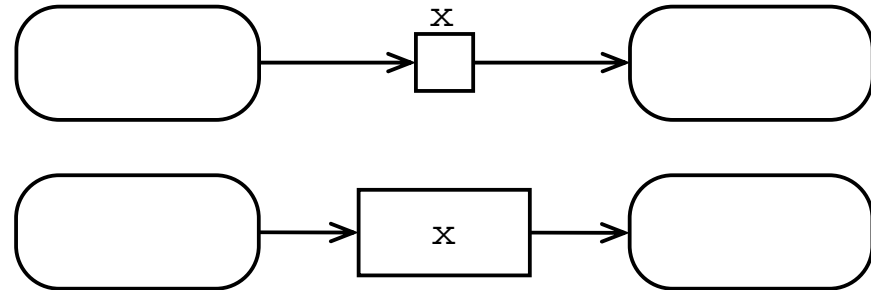
❑ Pins can specify streaming, i.e., tells whether an input pin may accept values while its behavior is executing, or whether an output pin post values while the behavior is executing.

{stream}   x   {stream}

p1   p2
{stream}   {stream}

p1   p2
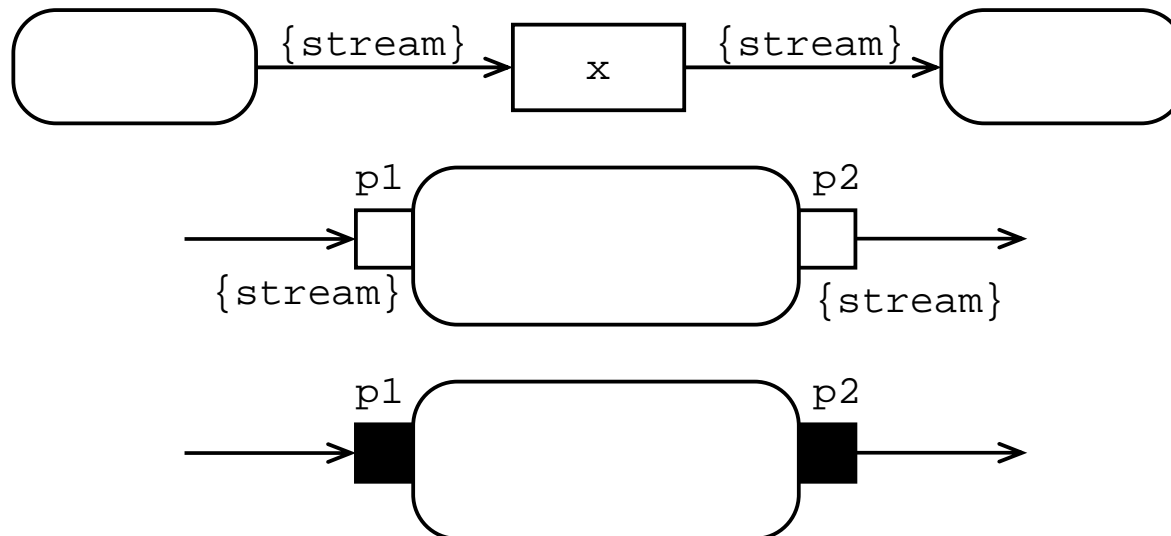
Object Flow

→ An activity edge that can have objects or data passing along it.

❏ Models the flow of values to or from object nodes.

❏ Object flows add support for multicast/receive, token selection from object nodes, and transformation of tokens.

❏ Object flows may not have actions at either end.

❏ Object nodes connected by an object flow must have compatible types and upper bounds.

Two object flows linking an object node and actions:
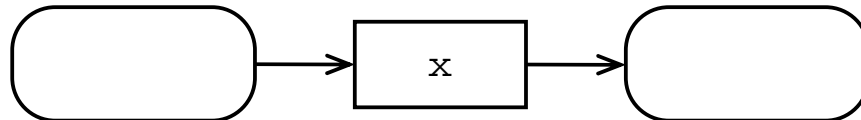
An object flow with selection linking two pins:

«selection»
selection specification
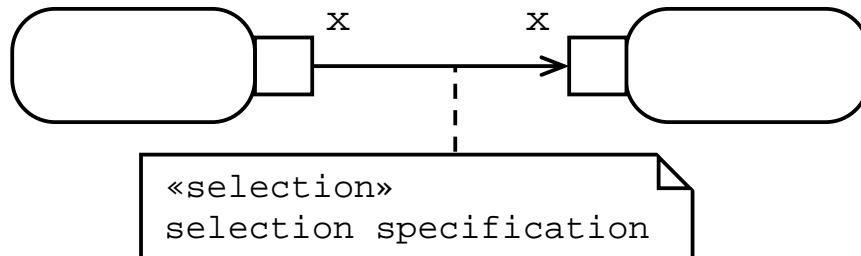
An object flow with pins elided:
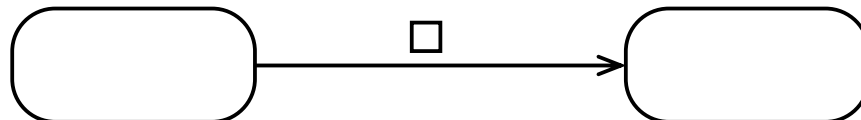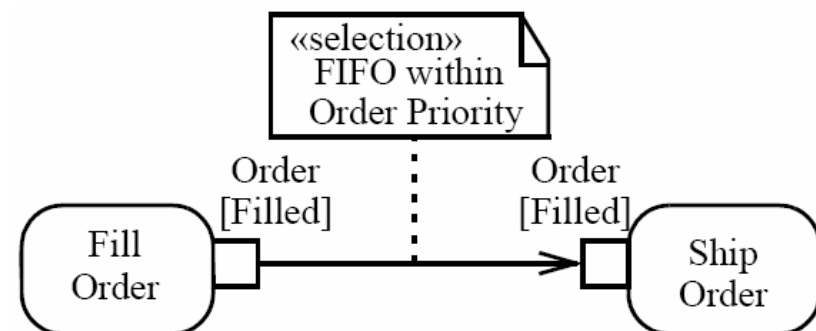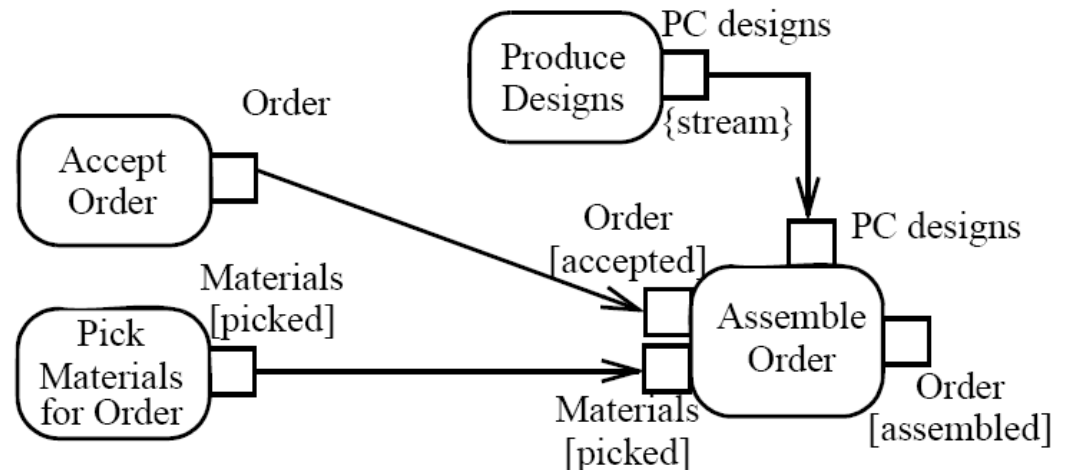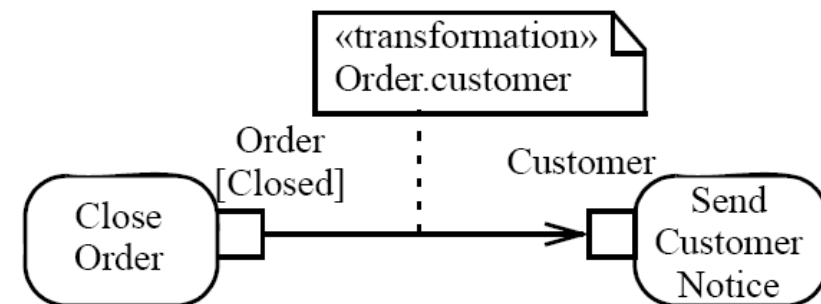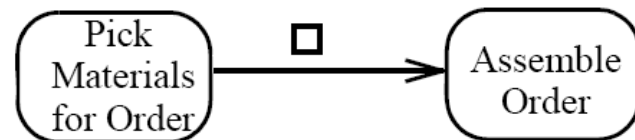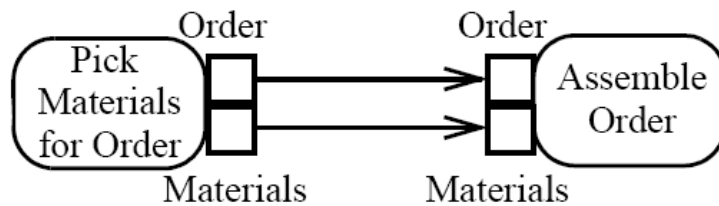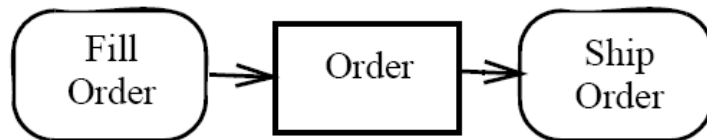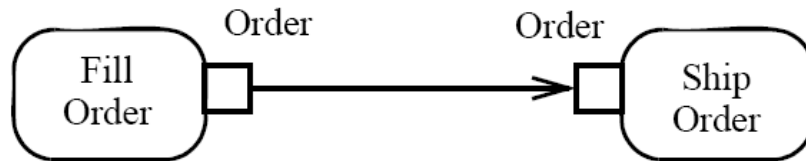
Decision Node

→ A control node that chooses between outgoing flows.

❑ Has one incoming edge and multiple outgoing activity edges.

❑ Each token arriving at a decision node can traverse only one outgoing edge.

❑ Guards of the outgoing edges are evaluated to determine which edge should be traversed.

- The evaluation order is not defined.

- The predefined 'else' guard can be used.

❑ A decision behavior/condition applied for each token before it is offered to the outgoing edges can also be specified.

«decisionInput»
decision behavior
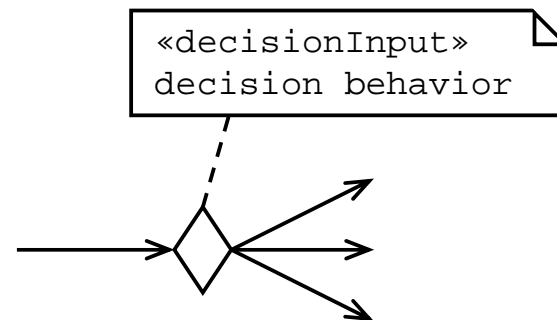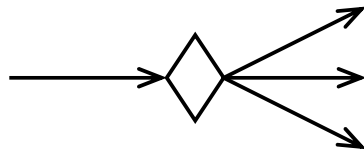
Merge Node

→ A control node that brings together multiple alternate flows.

❏ Has multiple incoming edges and a single outgoing edge.

❏ All tokens offered on incoming edges are offered to the outgoing edge.

- There is no synchronization of flows or joining of tokens.

merge and decision
nodes together
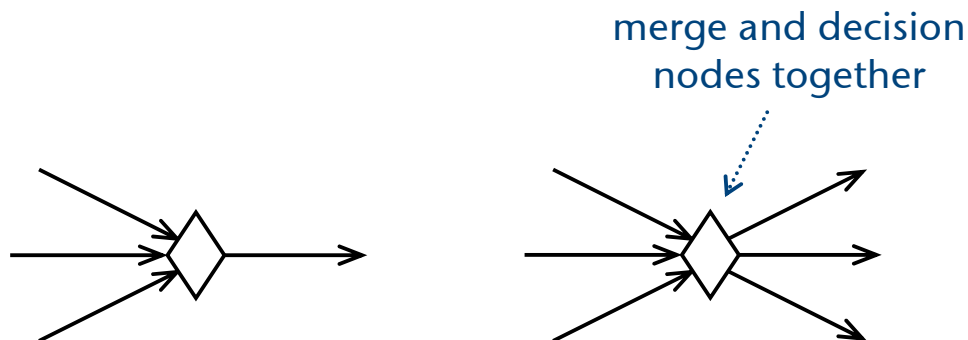
Examples of Decisions and Merges (2)
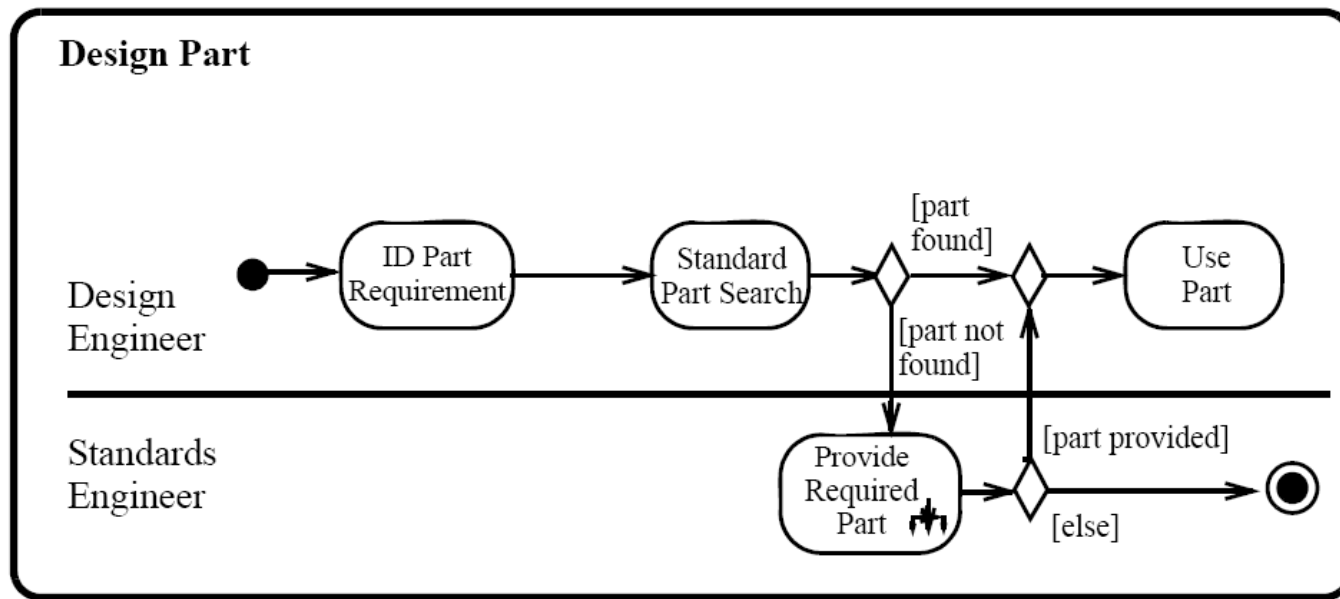
Fork Node

→   A control node that splits a flow into multiple concurrent flows.

❑   Has one incoming edge and multiple outgoing edges.

❑   Tokens arriving at a fork are duplicated across the outgoing edges.

- If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token.

- The outgoing edges that did not accept the token due to failure of their targets to accept it, keep its copy in an implicit FIFO queue until it can be accepted by the target.

- The rest of the outgoing edges do not receive a token (these are the ones with failing guards).
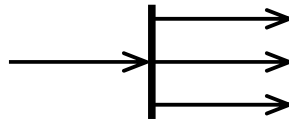
Join Node

→ A control node that synchronizes multiple flows.

❑ Has multiple incoming edges and one outgoing edge.

❑ Can have a boolean value specification (modeled by the *joinSpec* tag) using the names of the incoming edges to specify the conditions under which the join will emit a token. If the joinSpec is not given, then:

❑ If all the tokens offered on the incoming edges are control tokens, then one control token is offered on the outgoing edge.

❑ If some of the tokens offered on the incoming edges are control tokens and others are data tokens, then only the data tokens are offered on the outgoing edge. Tokens are offered on the outgoing edge in the same order they were offered to the join.

fork and join nodes together

`{joinSpec=…}`
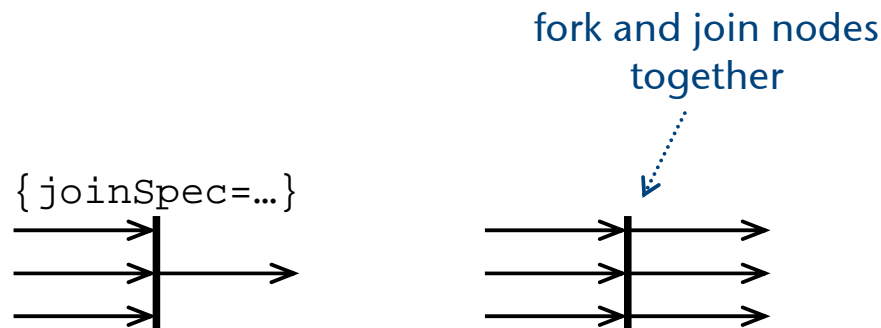
Examples of Fork and Join Nodes
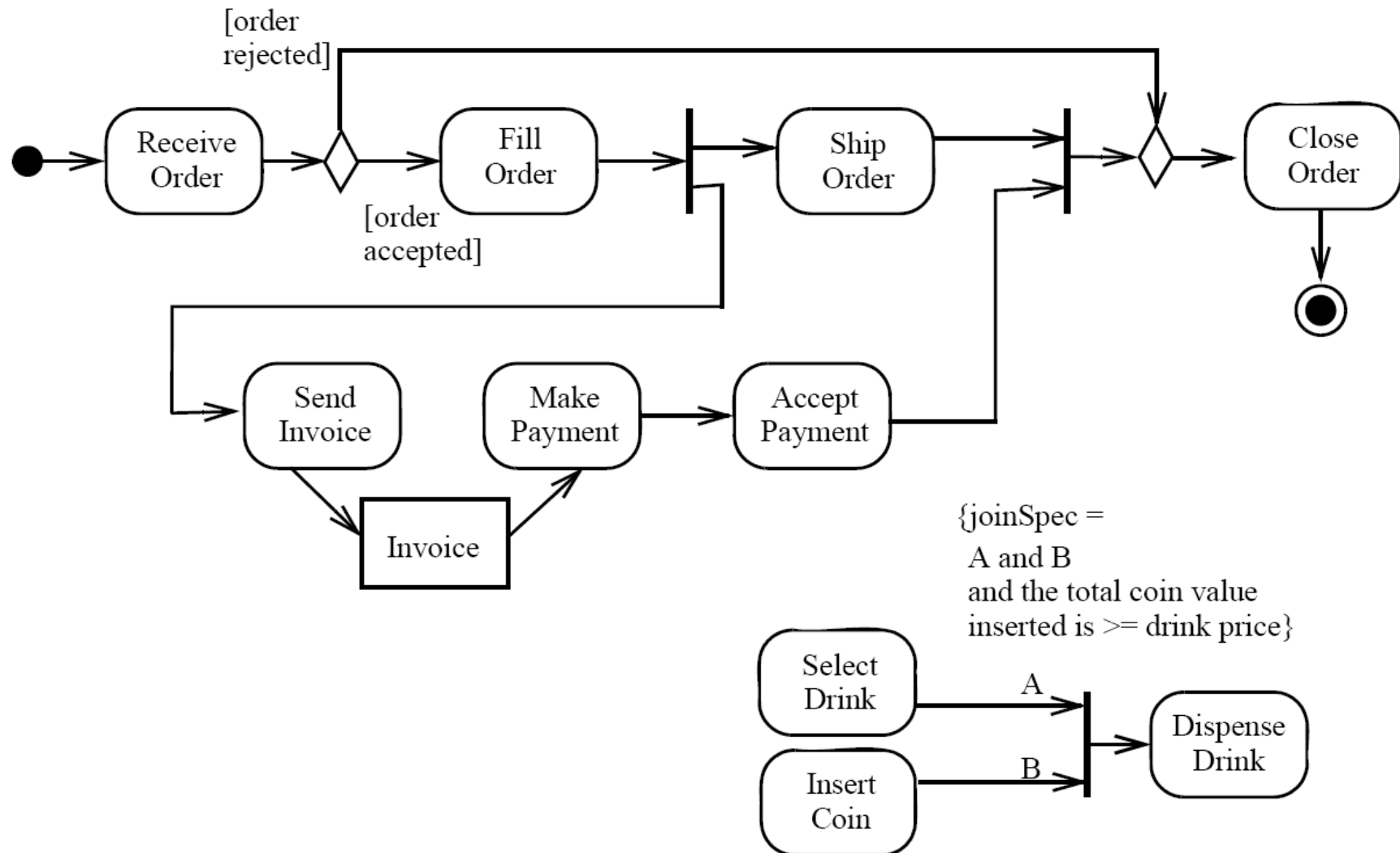
Exception Handler

→ Specifies a *body* to execute in case the specified *exception* occurs during the execution of the *protected node*.

❏ If an exception occurs (a Raise Exception Action is executed) in the protected node, all the tokens in the protected node are terminated. Then, the exception handlers are examined for matching the exception type, and the handler body of any matching exception handler is used to handle the exception which arrives via the exception input pin.

❏ If the exception is not caught at the level of the protected node, the exception handling process repeats at the level of the enclosing structured node or activity.

❏ If the exception is not caught at the top-most level of asynchronously invoked activity, the exception is lost.

❏ If the action that invoked the activity is synchronous, the exception propagates up to that action. The process of exception propagation recurs until the exception is caught, or reaches the topmost level of the system, where the behavior for the uncaught exceptions is unspecified.

❏ The result tokens of the handler body become the result tokens of the protected node.

Exception Handler (cont.)
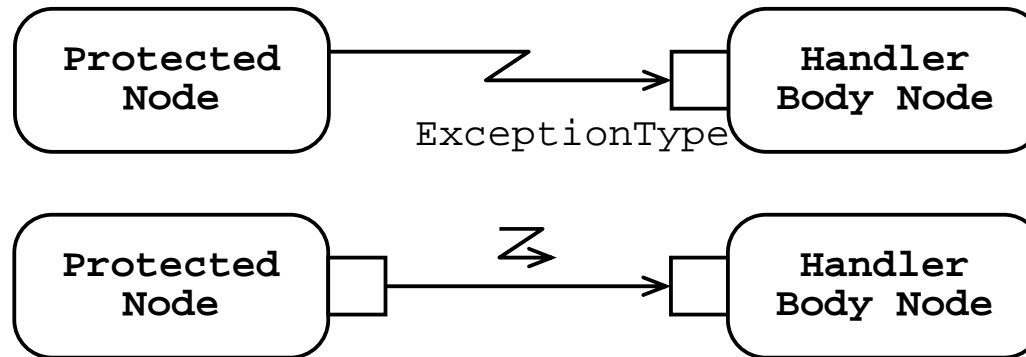
An alternative notation for exception flows and exception object nodes (activity parameters and pins) with a small triangle icon:
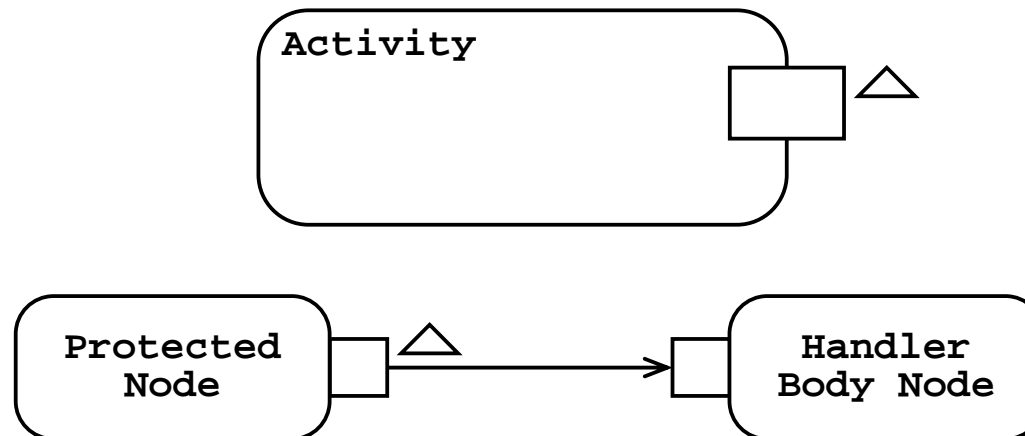
Example of Exception Handler
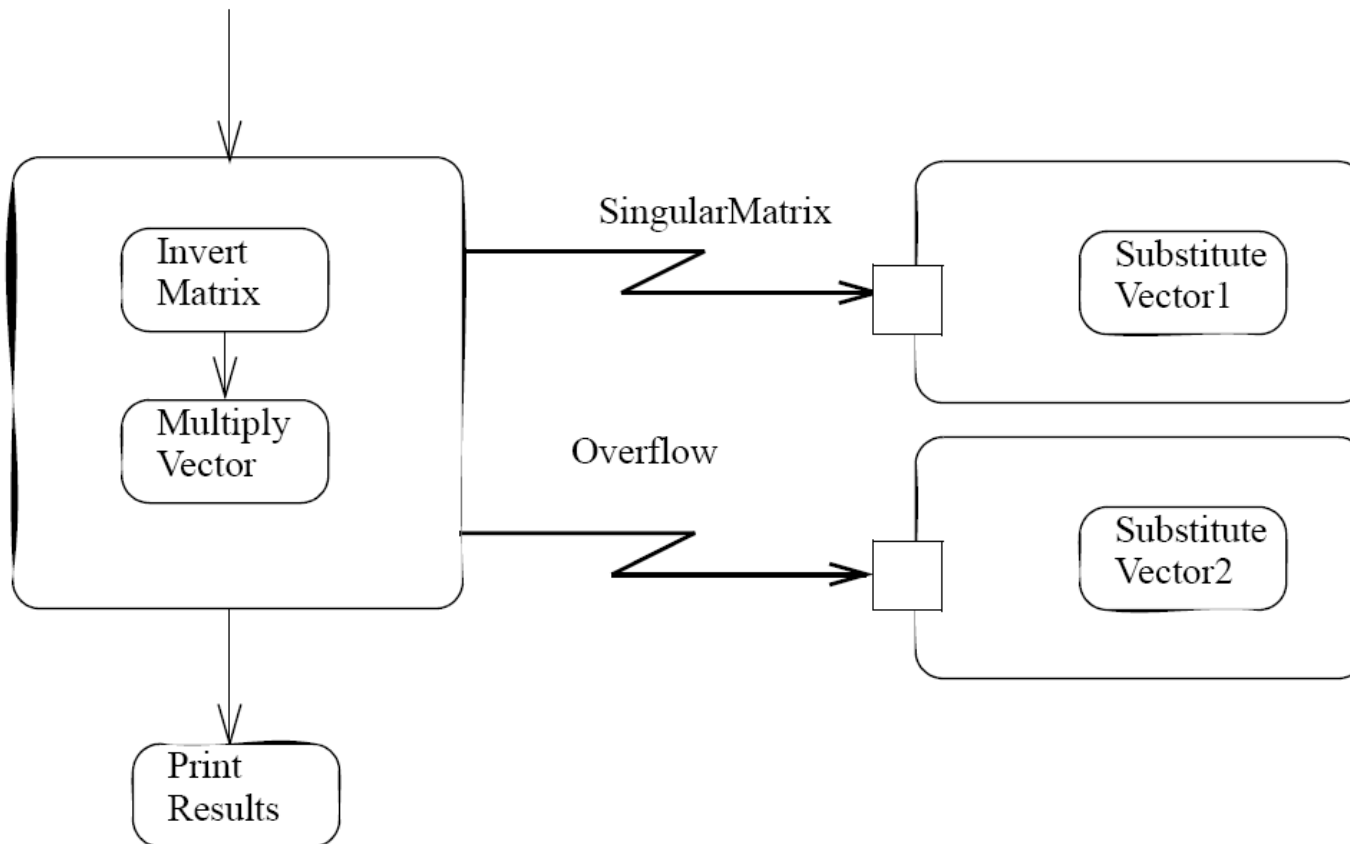
Interruptible Activity Region

→ An activity group that supports termination of tokens flowing in the portions of an activity.

❏ Contains other activity nodes.

❏ When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviors in the region are terminated.

❏ Interrupting edges of a region must have their source node in the region and their target node outside the region in the same activity containing the region.
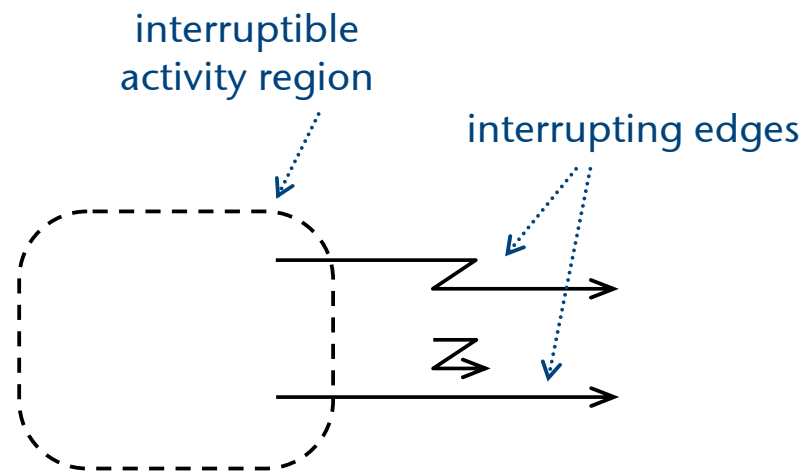
Example of Interrupting Activity Region
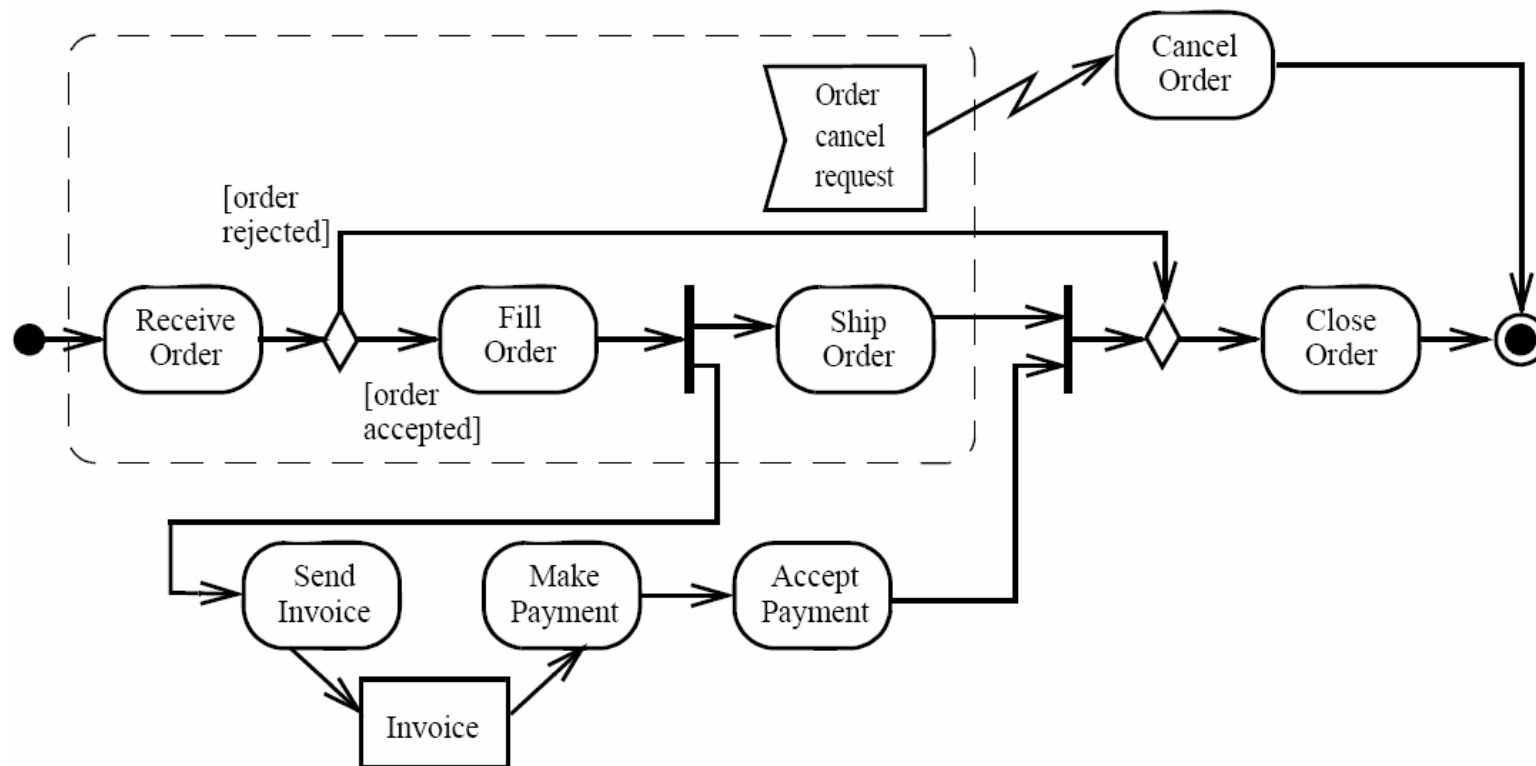
Central Buffer Node

→ An object node for managing flows from multiple sources and destinations.

❑ Accepts tokens from upstream object nodes and passes them along to downstream object nodes.
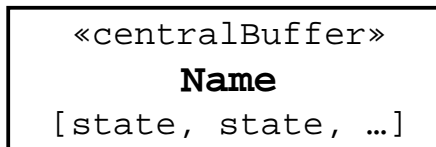
```
┌─────────────────────┐
│    «centralBuffer»  │
│        Name         │
│   [state, state, …] │
└─────────────────────┘
```
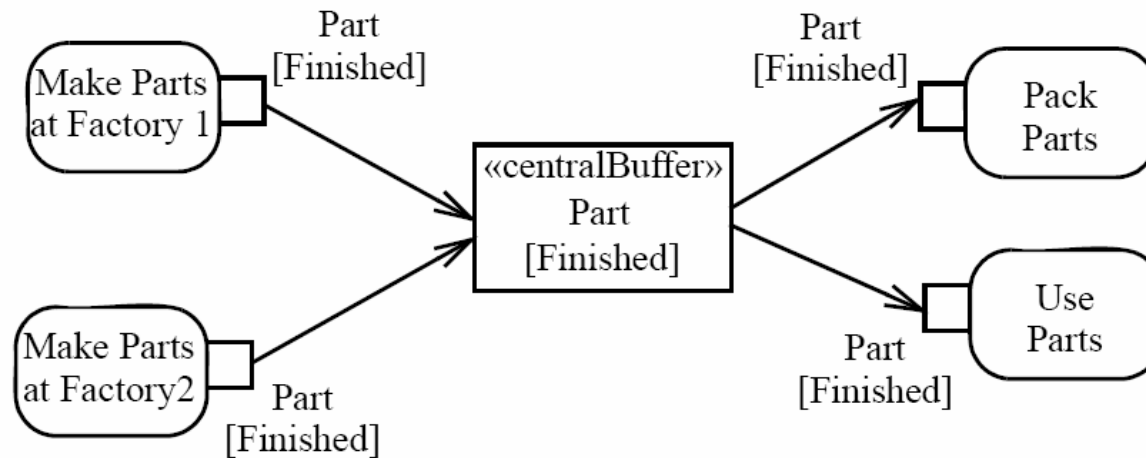
Data Store Node

→ A central buffer node for non-transient information.

❑ Keeps all tokens that enter it and copies them when they are chosen to move downstream.

❑ Selection and transformation behavior on outgoing edges can be designed to get information out of the data store, as if a query were being performed.

❑ Incoming tokens containing a particular object replace any tokens in the object node containing that object.

```
        «datastore»
          Name
    [state, state, …]
```
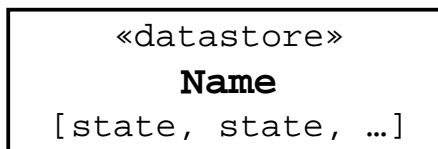
Example of Data Store Node
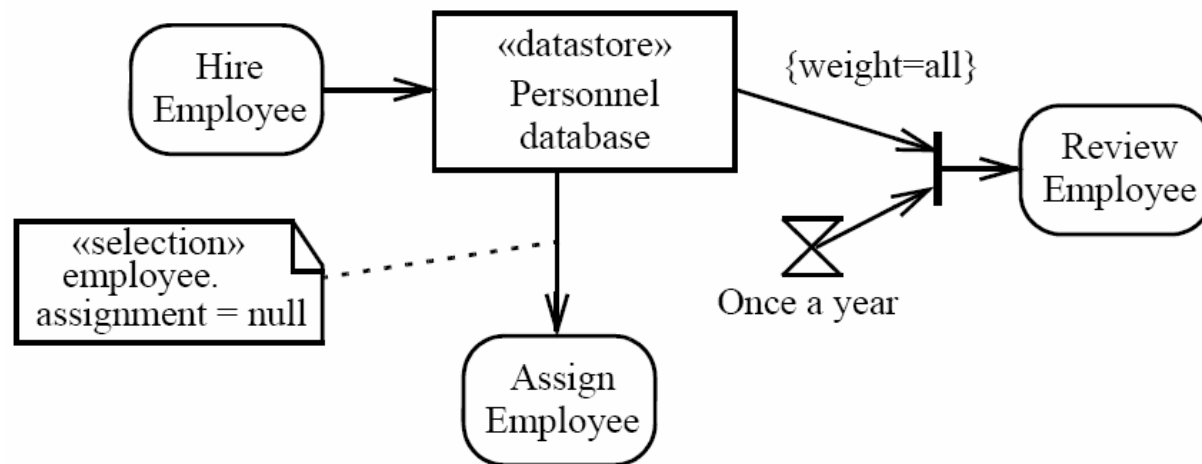
Expansion Region

Expansion Region

→ A structured activity region that executes multiple times corresponding to elements of an *input collection*.

❑ Each input is a collection of values modeled as an ***expansion node***.

- If there are multiple inputs to one expansion node, each of them must hold the same kind of collection.

- Each input flow edge produces elements of the input collection.

❑ The expansion region is executed once for each element (or position) in the input collection.

❑ On each execution of the region, an output value from the region is inserted into an *output collection*, modeled also as an expansion node, at the same position as the input elements.

- If the region execution ends with no output, then nothing is added to the output collection.

❑ From the inside of the region, expansion nodes are visible as individual values.

Expansion Region (cont.)

❑ Any object flow edges that cross the boundary of the region, without passing through expansion nodes, provide values that are fixed within the different executions of the region input pins.

❑ The *expansion kind* specifies the way in which the executions interact:

- *parallel*–all interactions are independent

- *iterative* (default)–the interactions occur in order of the elements

- *stream*–a stream of values flows into a single execution

input expansion node

expansion kind

output expansion node
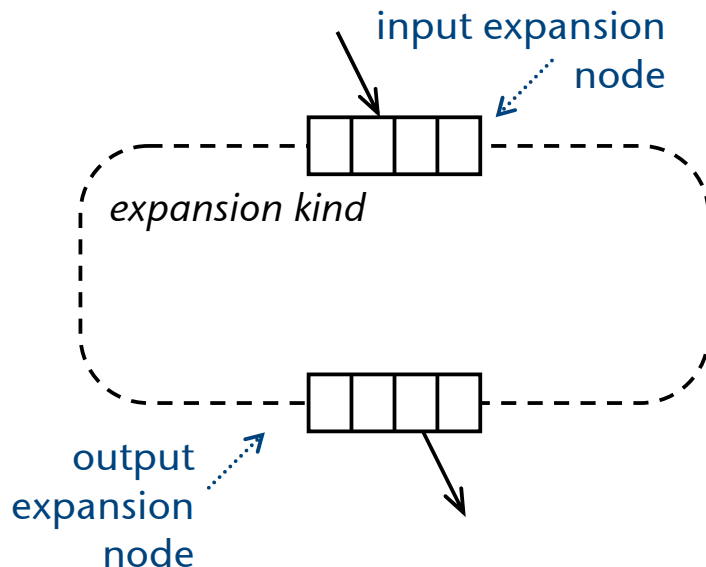
Shorthand notation for expansion region containing a single action:

expansion kind
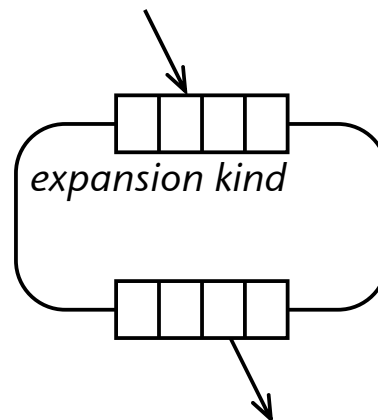
action

Shorthand notation for parallel expansion region containing a single action:

*

«parallel»

S: Array<Complex>

Slower, Supper = cut(S)

Slower: Array<Complex>

Supper: Array<Complex>

V: Array<Complex>

lower:Complex

upper:Complex

root:Complex

«parallel»

nxteven = lower+upper

nxtodd = (lower-upper)*root

nxteven:Complex

nxtodd:Complex

Sneven: Array<Complex>

Snodd: Array<Complex>

S = shuffle(Sneven,Snodd)

S': Array<Complex>

Specify Trip Route

Book Flight

Book Hotel

Print Itinerary

Specify Trip Route

Book Flight *
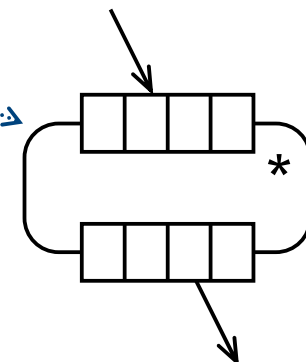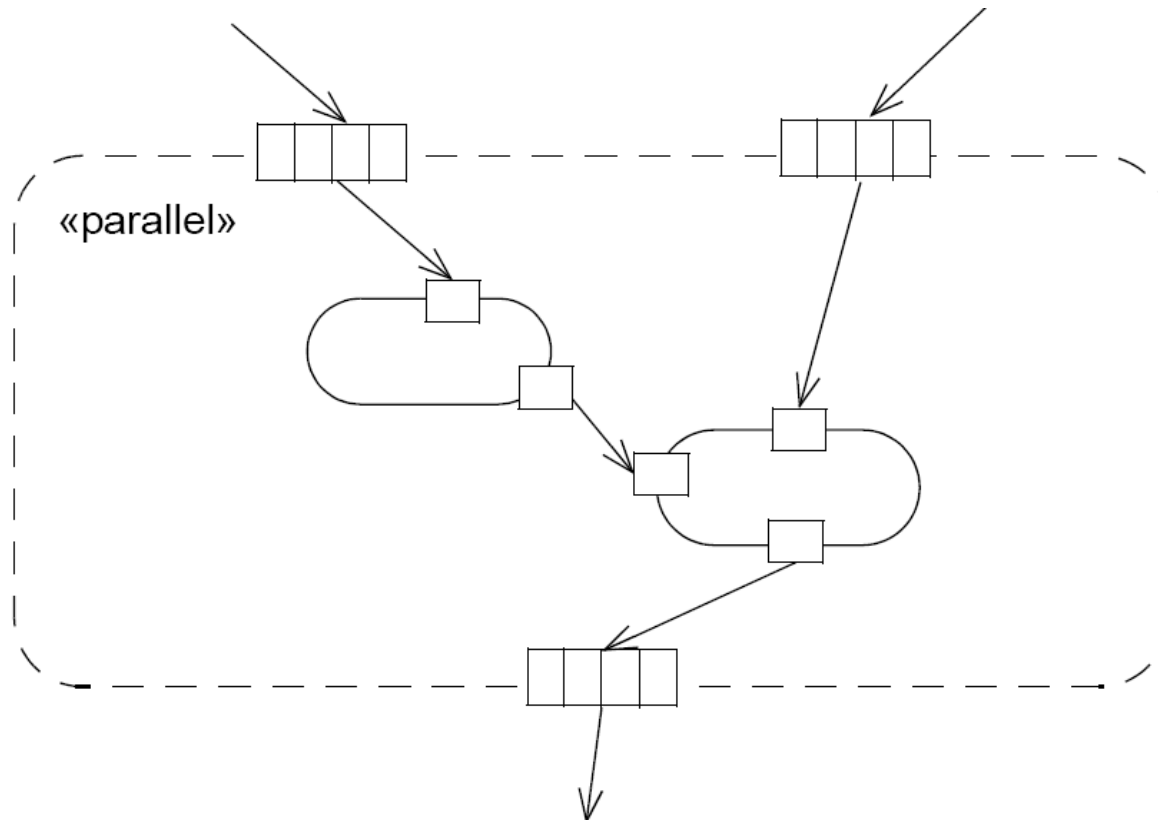
Activity Partition
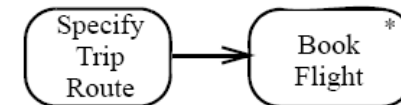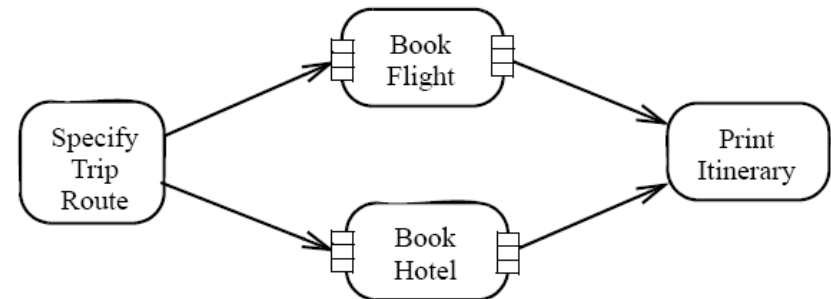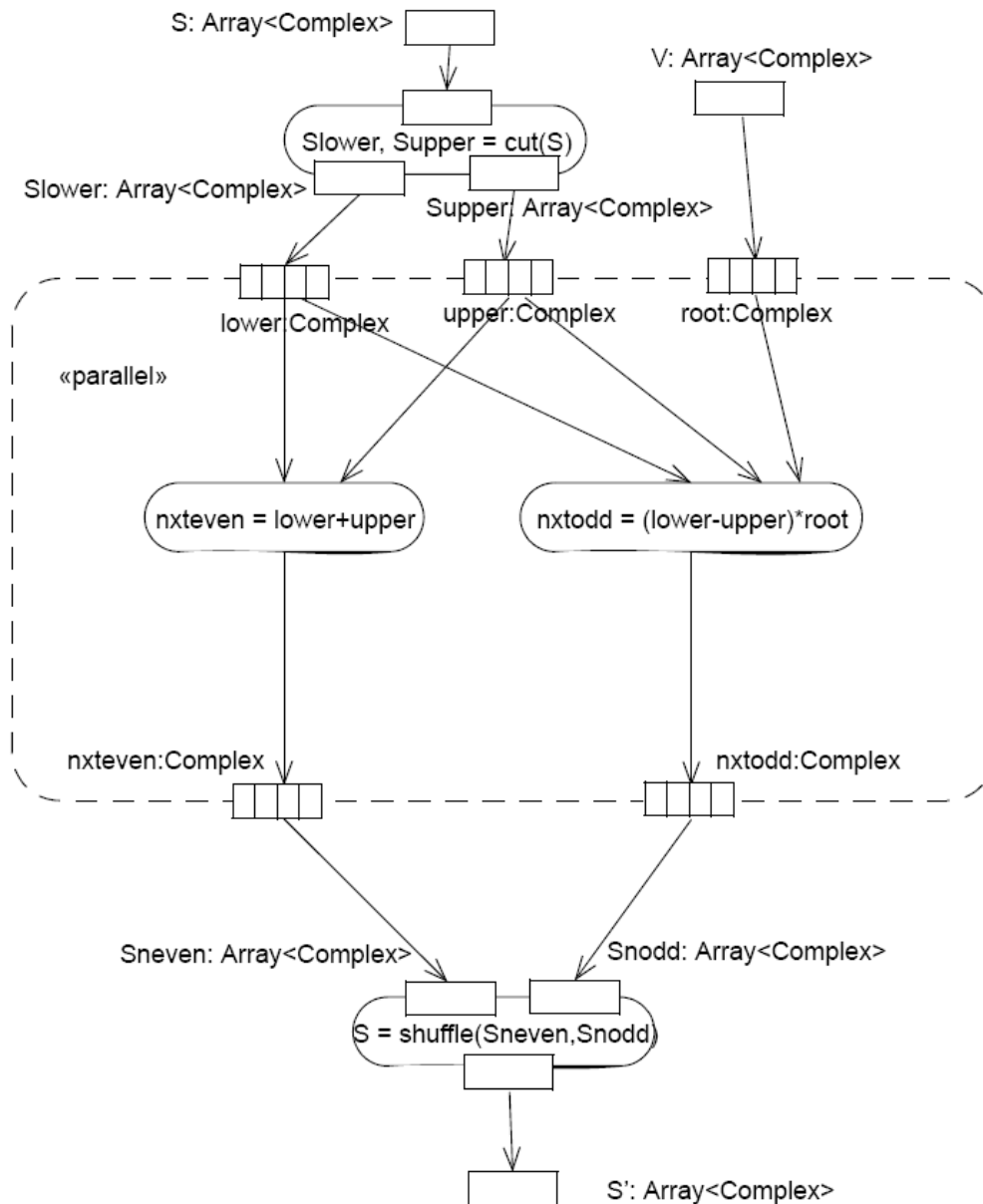
→ A kind of activity group for identifying actions that have some characteristic in common.

❑ A partition can represent a classifier, instance, part, attribute or value which is responsible for execution of the contained activity part.

❑ An **external partition** (marked by *«external»*) represents an entity to which the partitioning structure does not apply.

❑ Partitions can share contents.

❑ Partitions can be hierarchical and multi-dimensional.

❑ Partitions do not affect the token flow of the model.

Activity Partition (cont.)

Horizontal partition:
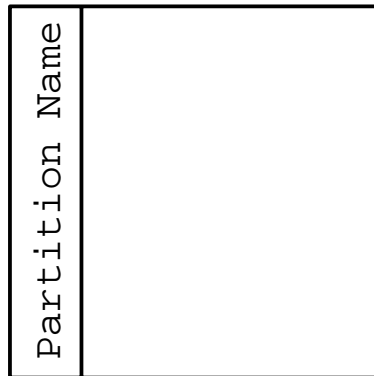
Partition Name	

Vertical partition:
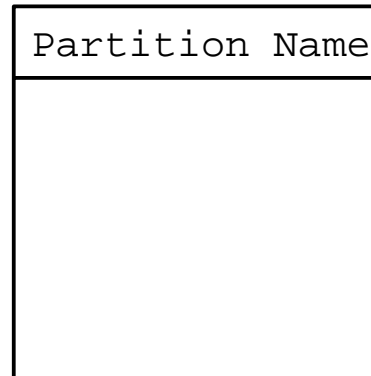
Partition Name

Hierarchical partitions:

Partition Name	
Sub-partition Name-1	Sub-partition Name-2
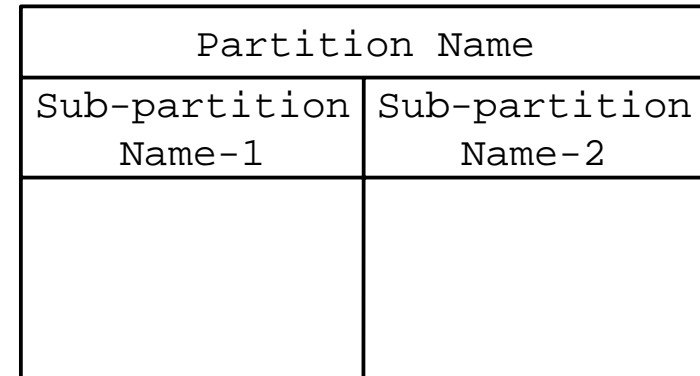
Multi-dimensional partitions:

	Partition Name-3	Partition Name-4
Partition Name-4		
Partition Name-3		
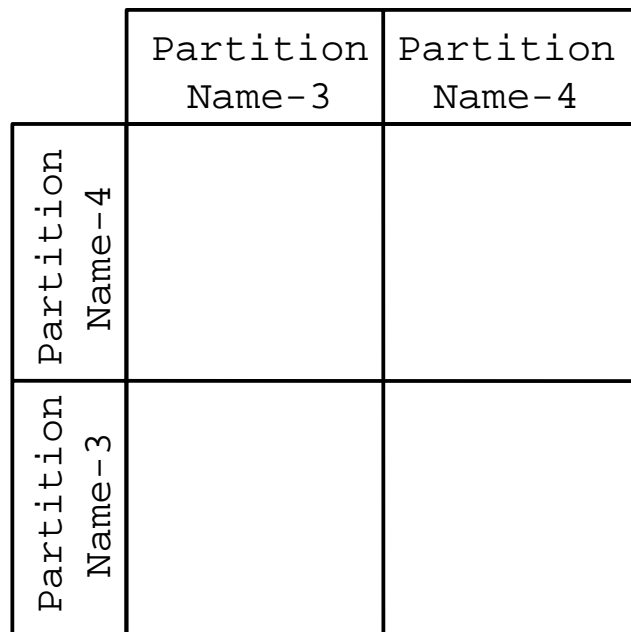
Partition(s) notated on a specific activity:

```
(Partition Name)
     Name
```

```
(Name:Sub-name)
     Name
```

```
(Name1, Name2,…)
      Name
```

```
  «external»
(Partition Name)
     Name
```
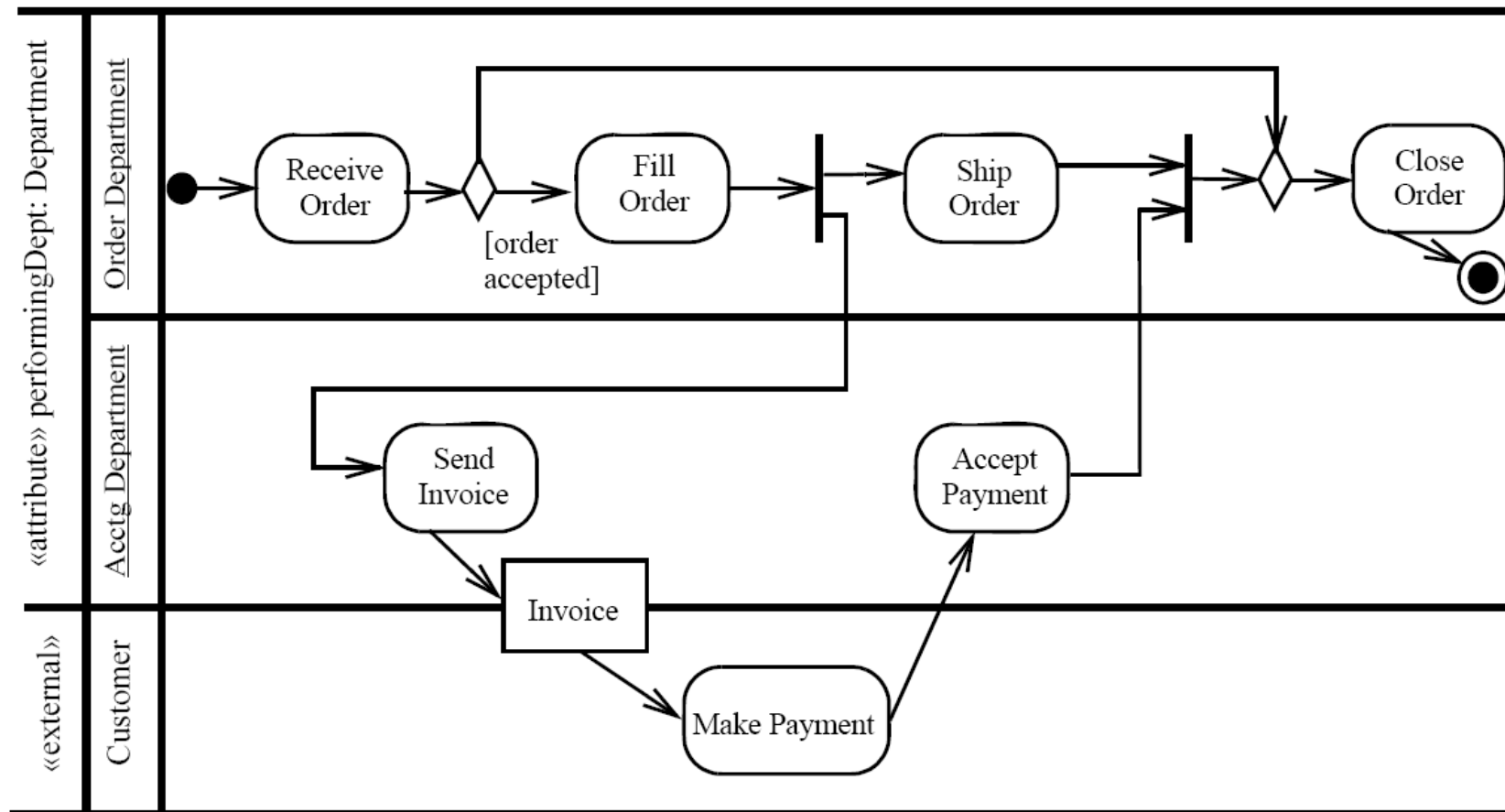
Examples of Activity Partitions (2)
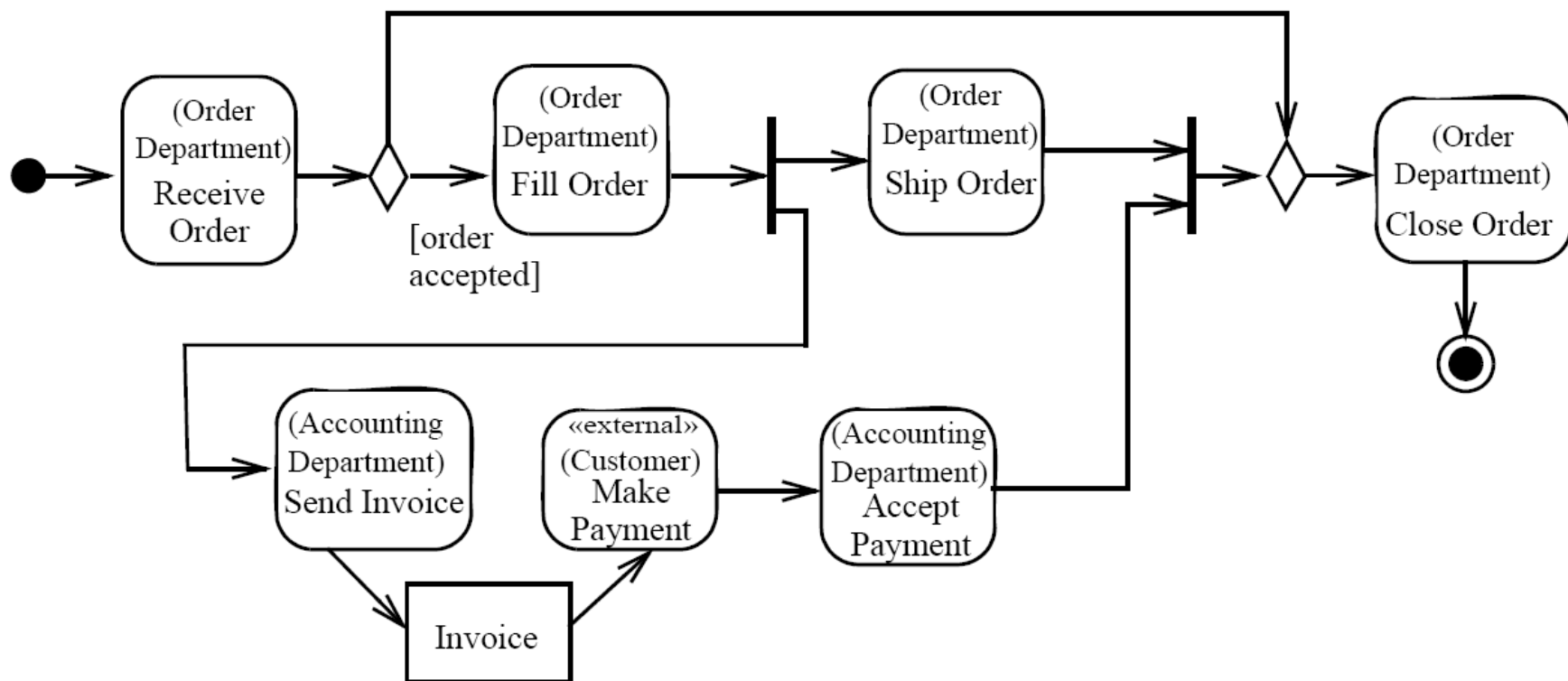
Examples of Activity Partitions (3)
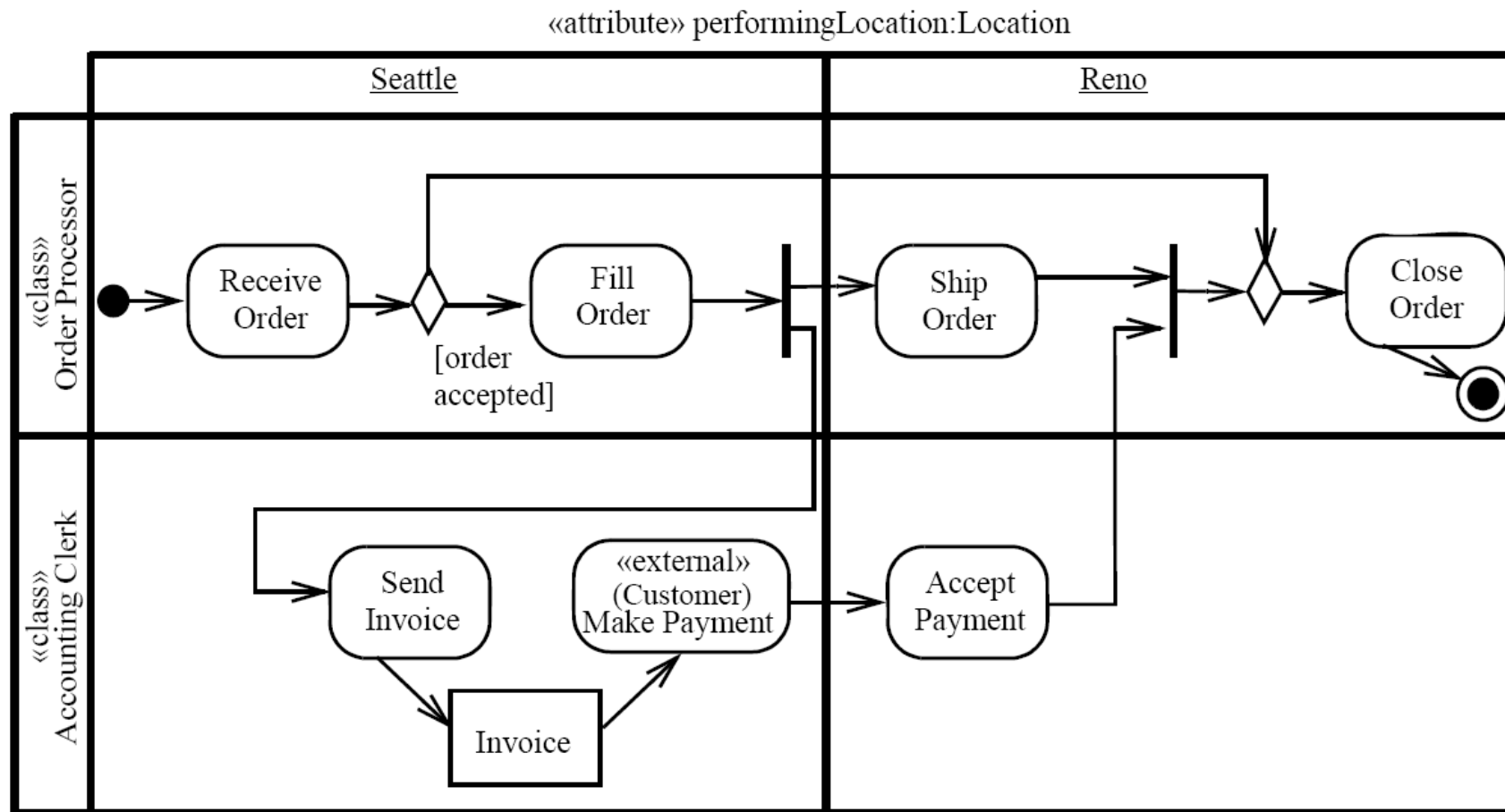
Object-Oriented Analysis and Modeling

State Machines

State Machine Model

→ **Used for modeling discrete behavior through finite state transition systems (called also final state machine - FSM).**

❏ *Behavioral state machines*—used to specify behavior (life cycle) of various model elements, e.g., classes, instances, subsystems, or components.

❏ *Protocol state machines*—used to express *usage protocols*, i.e., legal usage scenarios of classifiers, interfaces, or ports.

❏ The UML§s state machine formalism is an object-based variant of *Harel statecharts*.

Consists of:

- State machine diagrams.
- Element descriptions.

Used (mainly) in:

- Analysis and design ⇒ specification of behavior of various model elements, or specification of usage protocols.

State Machine

→ Specifies the behavior of part of a system in the form of final state automata.

❏ Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences.

❏ During this traversal, the state machine executes a series of activities associated with various elements of the state machine.

❏ A state machine owns one or more regions, which in turn own vertices and transitions. Each region determines an independent state machine which runs in parallel to the state machines in other regions.

❏ The *behaviored classifier* owning a state machine defines which signal and call triggers are defined for the state machine, and which attributes and operations are available in activities of the state machine.

- A state machine without a context classifier may use triggers that are independent of receptions or operations of a classifier.

❏ A state machine is drawn as a state machine diagram with its contained nodes and vertices. The content can optionally be placed within a diagram frame having the label "*stm*" before name.

Examples of State Machines (3)
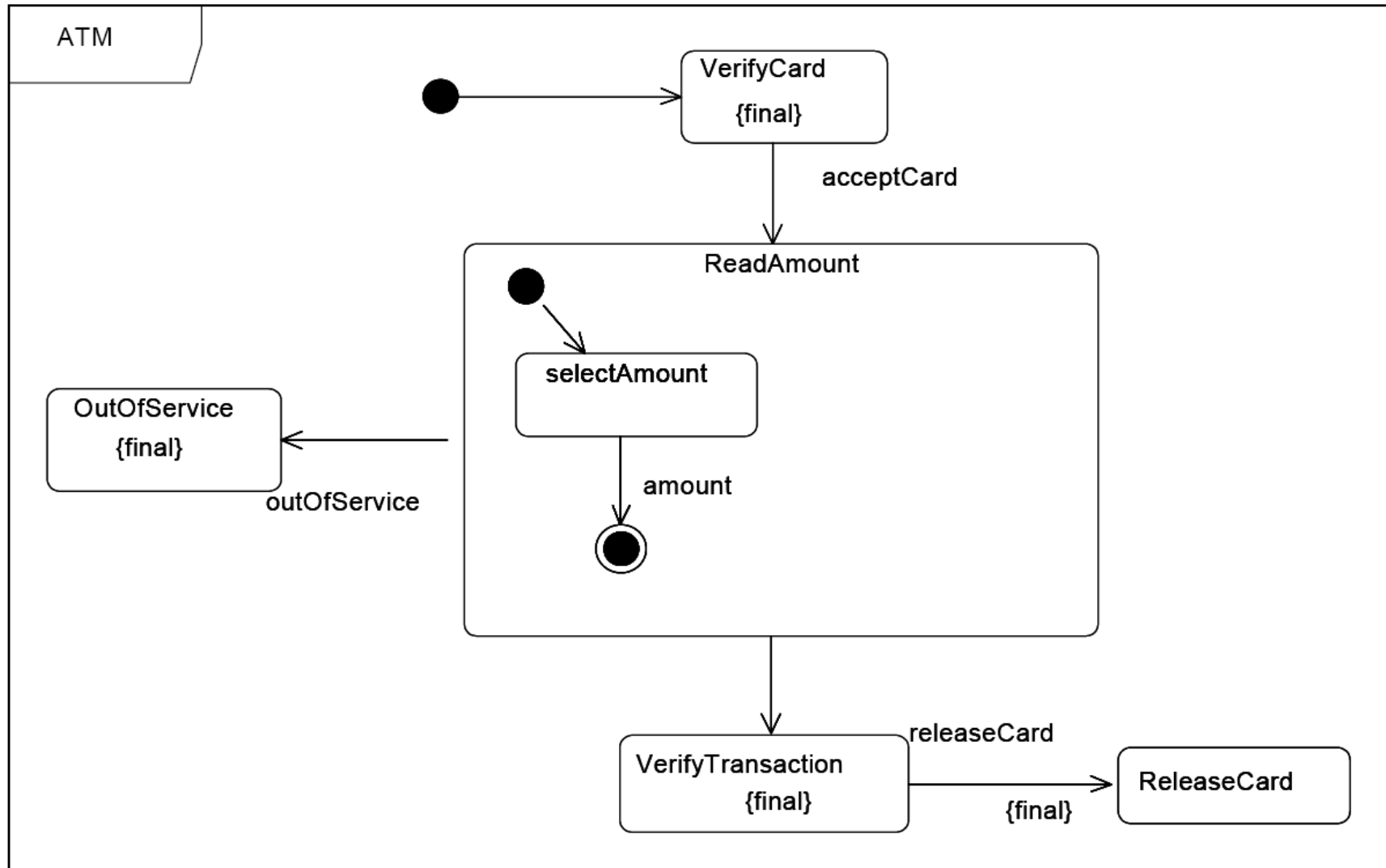
(Simple) State

→ Represents a situation during which some (usually implicit) invariant condition holds.

❑ The invariant may represent a static situation (such as an object waiting for some external event to occur) or a dynamic condition (such as the process of performing some behavior).

❑ A state becomes active when it is entered as a result of some transition, and becomes inactive if it is exited as a result of a transition.

❑ *Internal behaviors*—executed within the state.

- Format: *event '/' behavior-expression*
- Special events:
  - *entry*: entry to the state; specifies the *entry behavior*.
  - *exit*: exit from the state; specifies the *exit behavior*.
  - *do*: as long as the modeled element is in the state or until the computation specified by the expression is complete; specifies the *do activity* (behavior).

(Simple) State (cont.)

- ❒ *Internal transitions*—executed without exiting or re-entering the state in which they are defined.

  - Format as for transitions.

- ❒ *Deferrable events*—do not trigger any transitions in the current state, but remain in the event pool ready for processing by another state or transition.

  - Format: *event* '/' 'defer'.

```
 _____              ____
|        |            |Name|_____
|  Name  |            |____         ____|
|_____|            |                 |
                      |_____|
```

Name
internal behaviors
internal transitions
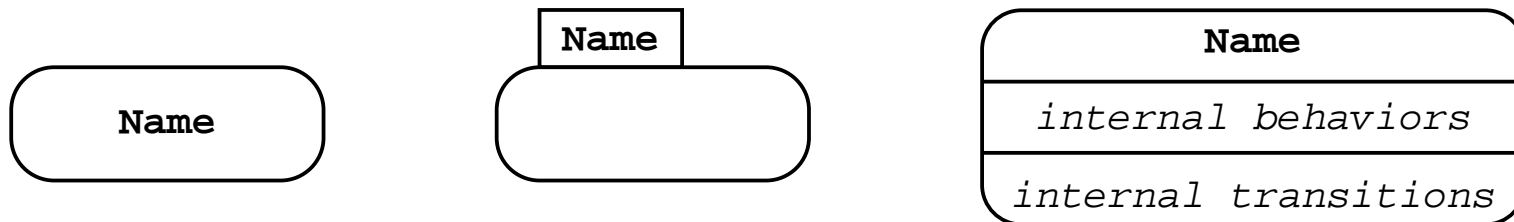
Example of Simple State

```
┌─────────────────────────────────────┐
│          TypingPassword             │
├─────────────────────────────────────┤
│ entry / setEchoInvisible            │
│ exit / setEchoNormal                │
│ keystroke / handleCharacter         │
│ help / displayHelp                  │
└─────────────────────────────────────┘
```

Composite State

→ A composite state either:

- contains one region (called ***non-orthogonal composite state***), or
- is decomposed into two or more orthogonal regions (called ***orthogonal composite state***).

❑ Each region can contain a set of mutually exclusive disjoint *substates* and a set of transitions.

❑ If the composite is active, each of its owned regions can have at most one immediately contained substate active. Therefore, if a state is active, all its transitively contained states are also active.

❑ A transition to the enclosing state represents a transition to the initial pseudostate in each region.

❑ A transition going directly to a substate activates it explicitly and all other orthogonal regions (if any) are activated in their initial pseudostates.

❑ A transition to a final state of the region represents the completion of behavior in the enclosing region.

❑ Completion of behavior in all orthogonal regions represents completion of behavior by the enclosing composite state.

Composite State (cont.)

❐ When exiting from a composite state, the active substates are exited recursively.

  • The exit behaviors are executed in sequence starting with the innermost active states in the current state configuration.

Non-orthogonal composite sate:

Orthogonal composite sate:
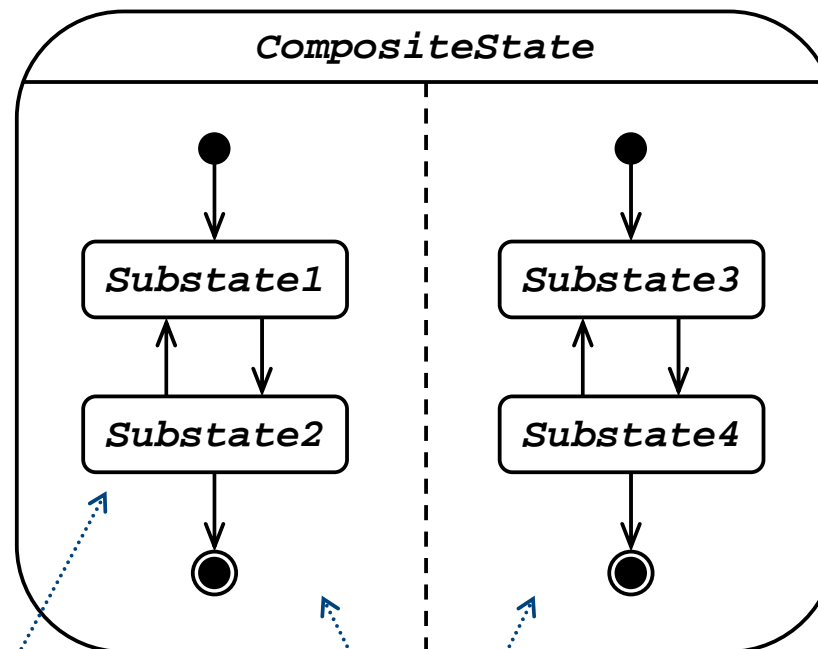
Composite sate with hidden decomposition:
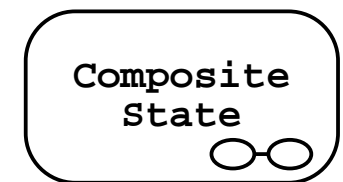
substates

regions
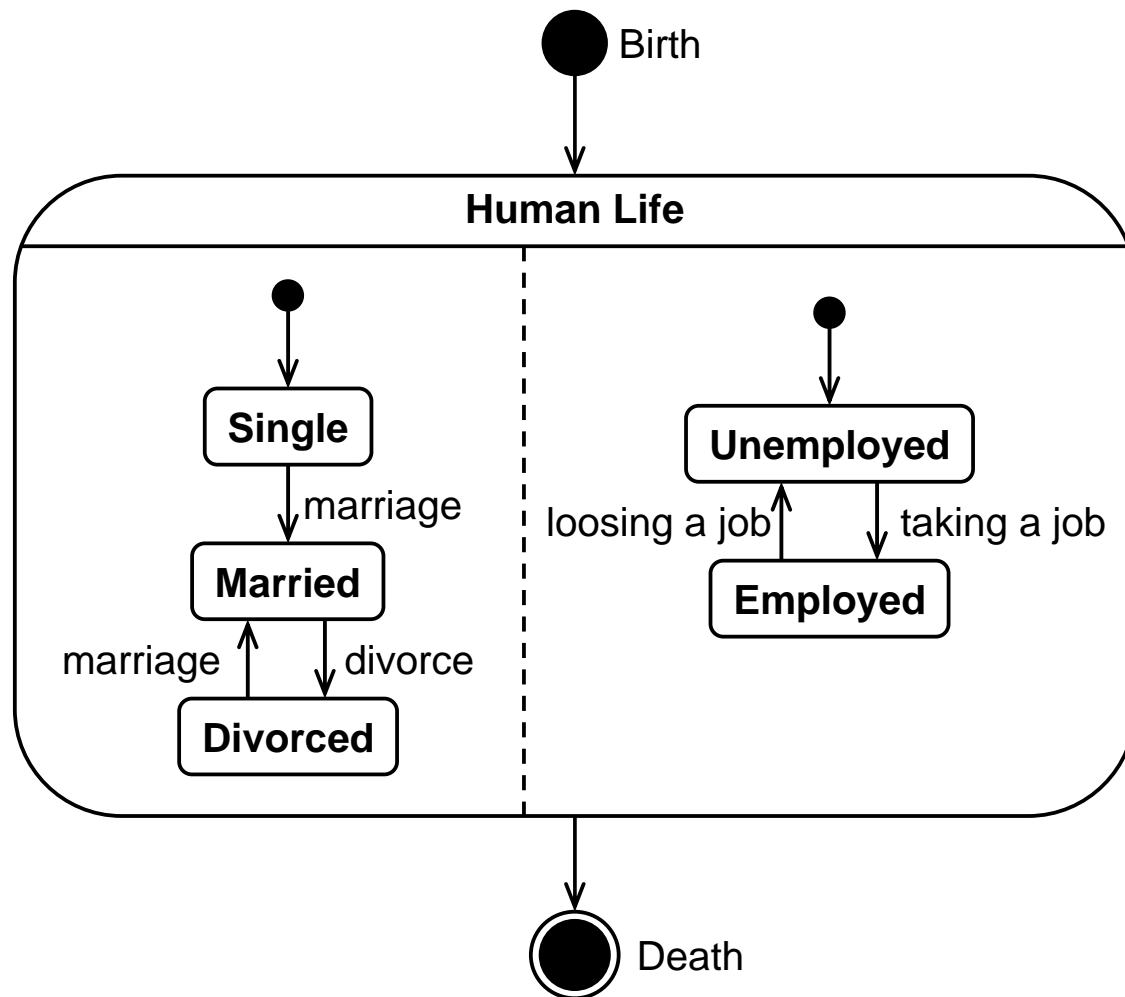
Examples of Composite States (2)
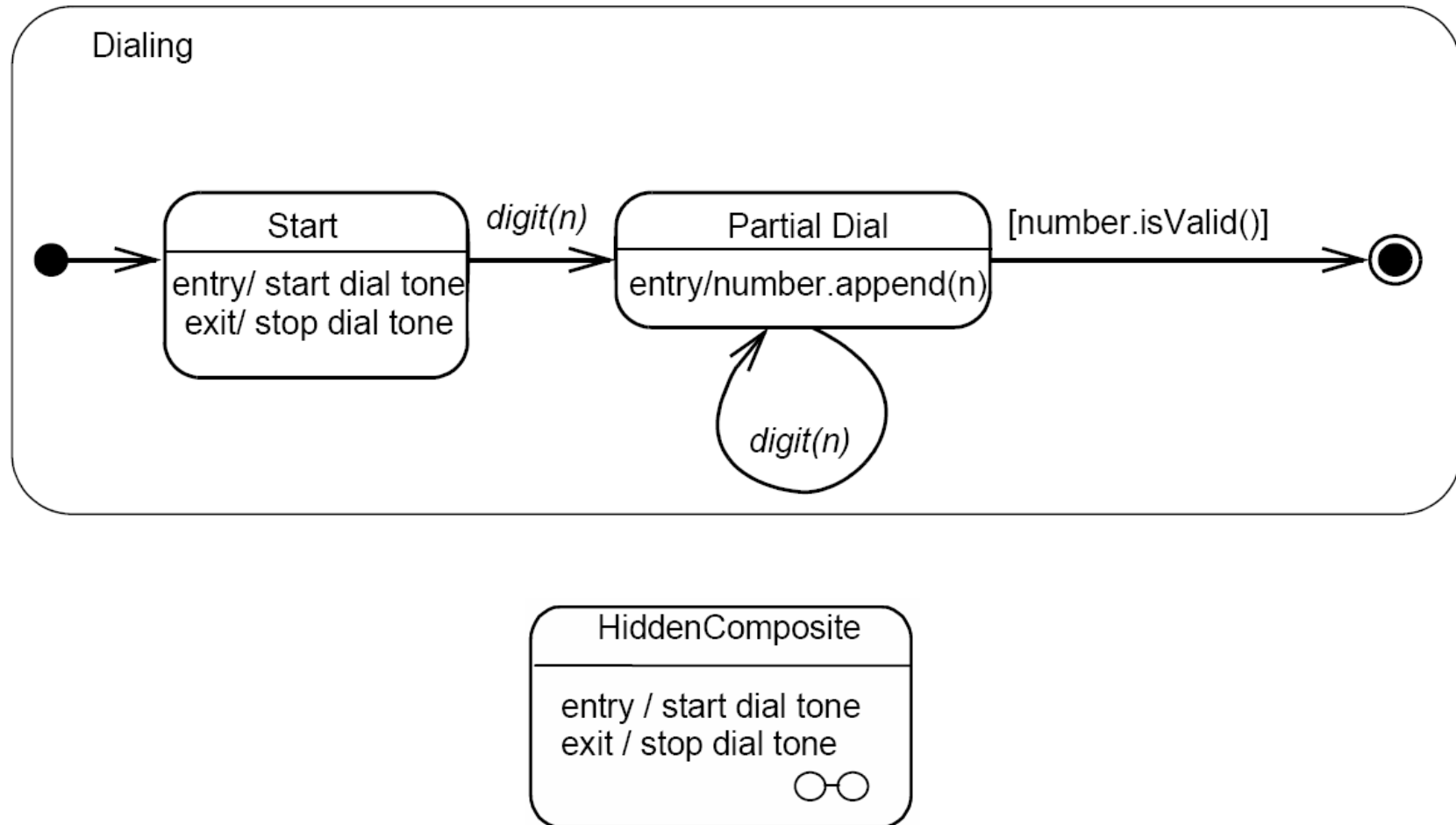
Examples of Composite States (3)
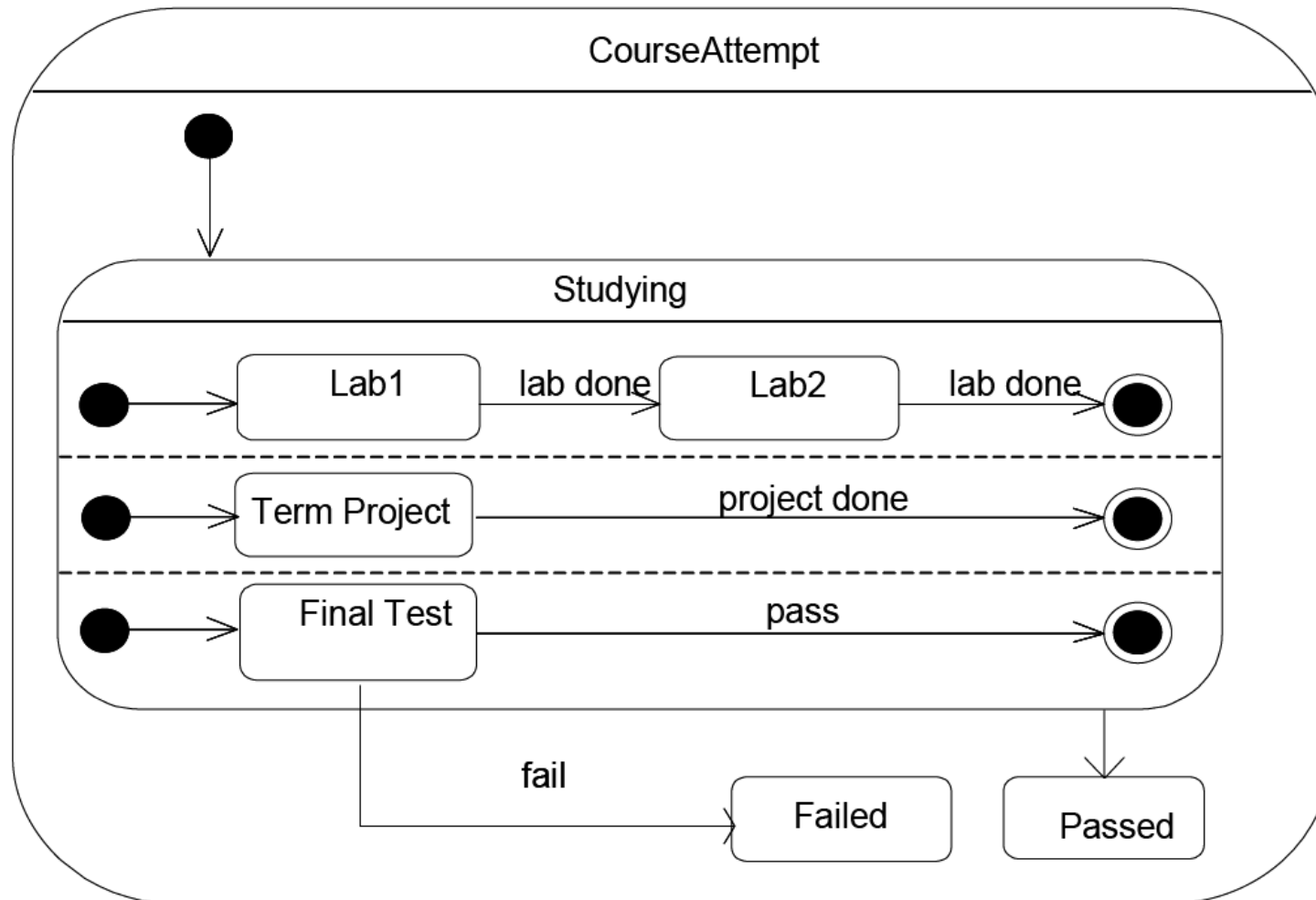
Submachine State

→ Specifies the insertion of the specification of a submachine state machine.

❑ Semantically equivalent to a composite state.

- Regions of the submachine state machine are the regions of the composite state.

- Transitions in the containing state machine can have entry/exit points of the inserted state machine as targets/sources.

❑ The same state machine may be a submachine more than once in the context of a single containing state machine.

```
Name:ReferencedStateMachine
```

Examples of Submachine States (1)
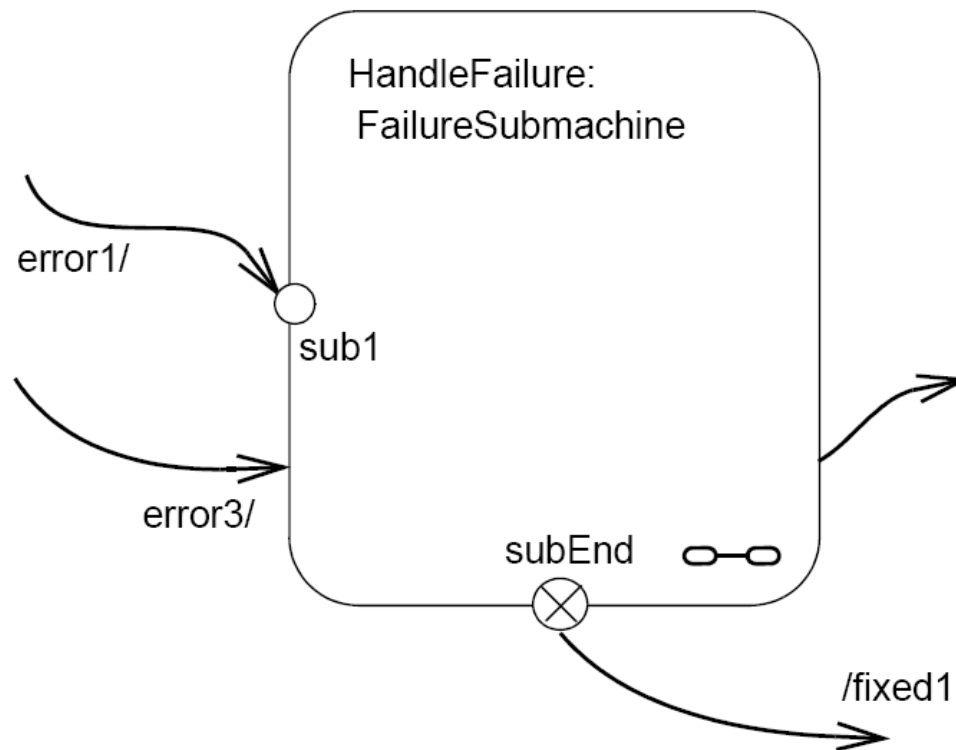
Examples of Submachine States (2)
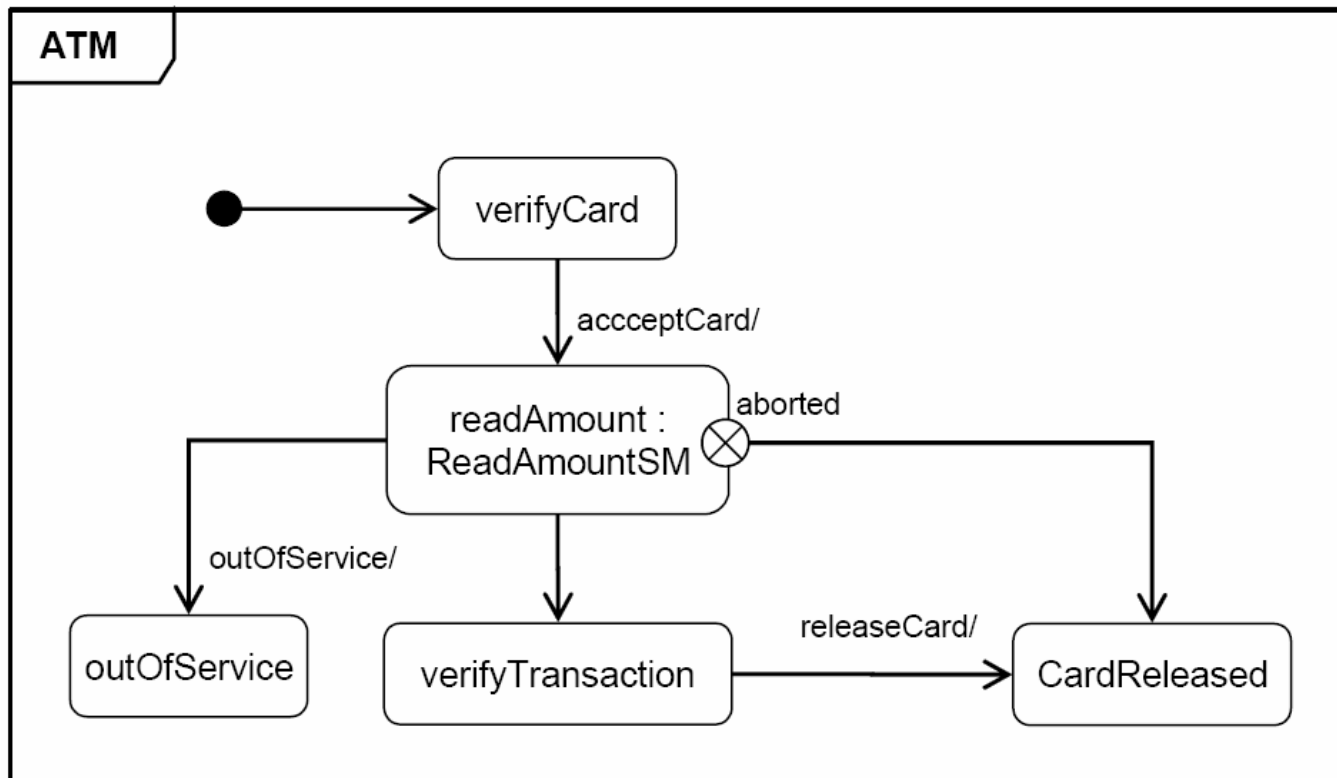
Examples of Submachine States (3)

Two alternatives of state machine referable from a submachine state:
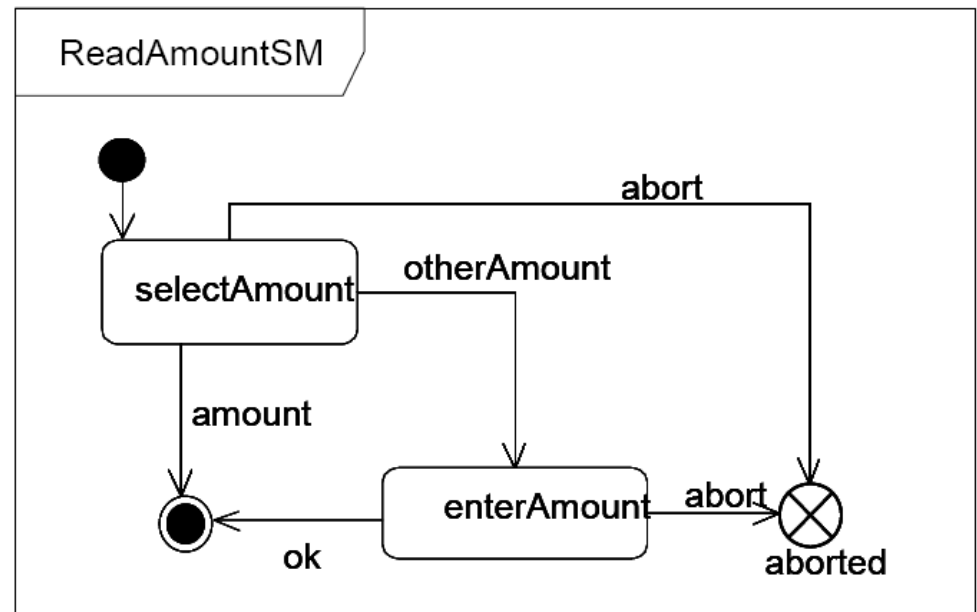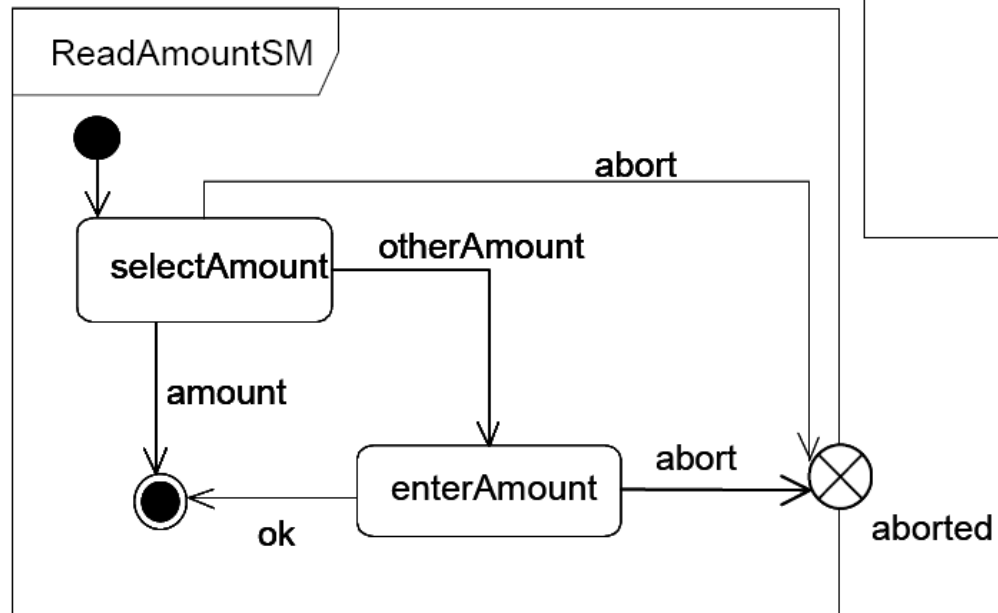
Transition

→ A directed relationship between a source vertex and a target vertex, which takes the state machine from one state configuration to another, representing the response to an occurrence of an event of a particular type.

❑ *High-level* (*group*) *transition*–originating from composite states.

   ● If triggered, they result in the innermost exiting of all the substates.

❑ *Compound transition*–a "semantically complete" an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates (see later) that define path from a set of source states to a set of destination states.

   ● A *simple transition* connecting two states is therefore a special common case of a compound transition.

❑ *Completion transition*–a transition originating from a state or an exit point which does not have an explicit trigger and is implicitly triggered by a completion of its source.
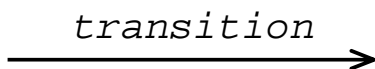
Simple transition: Compound transitions:

transition

[...]

[...]
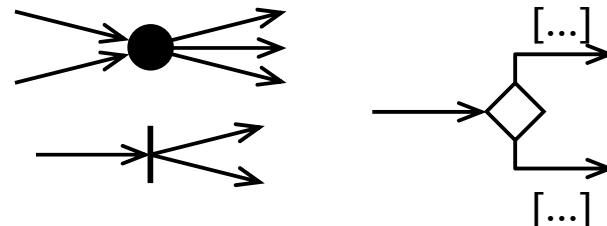
Transition (cont.)

❑ Format:

transition ::= [trigger [',' trigger]* ['[' guard ']'] ['/' behavior-expression]]

- *trigger* specifies the event which fires the transition

  trigger ::= name ['(' [attr-spec [',' attr-spec]*] ')'] |
  'all' | ('after' | 'at' | 'when') time-expression

  attr-spec ::= attr-name [':' type-name]

  - *attr-spec* specifies the event attribute; its name and type

  - 'all' determines the accepting of all events

  - 'after', 'at' and 'when' determine the time event relative to the specified time

- *guard* specifies a condition which enables firing the transition

- *behavior-expression* specifies the behavior which is executed by firing the transition

❑ Example:

```
right-mouse-down(location) [location in window] /
    object := pick-object(location);object.highlight()
```

Transition–Presentation Options

□ Trigger = signal receipt
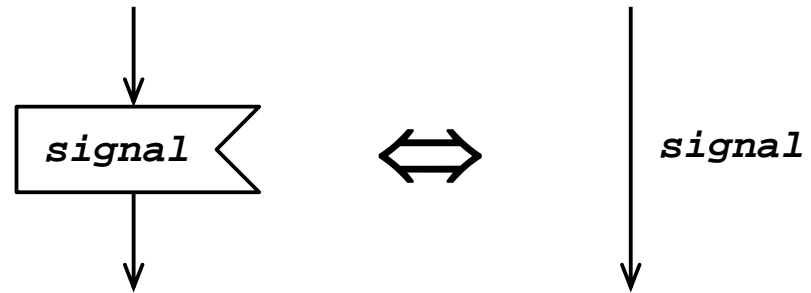
□ Behavior = send signal action
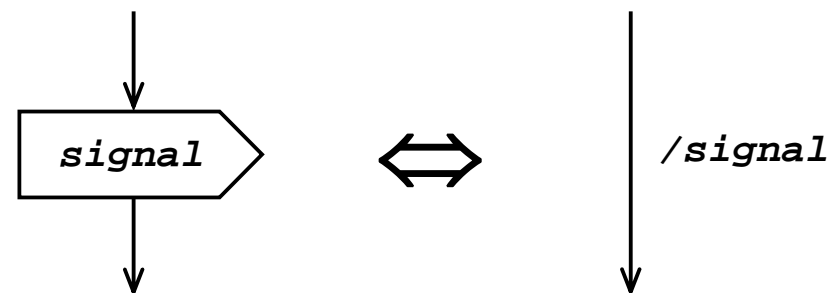
□ Behavior = other action
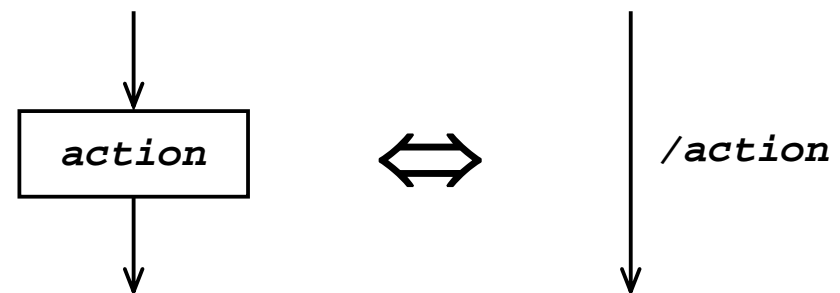
Examples of Transitions
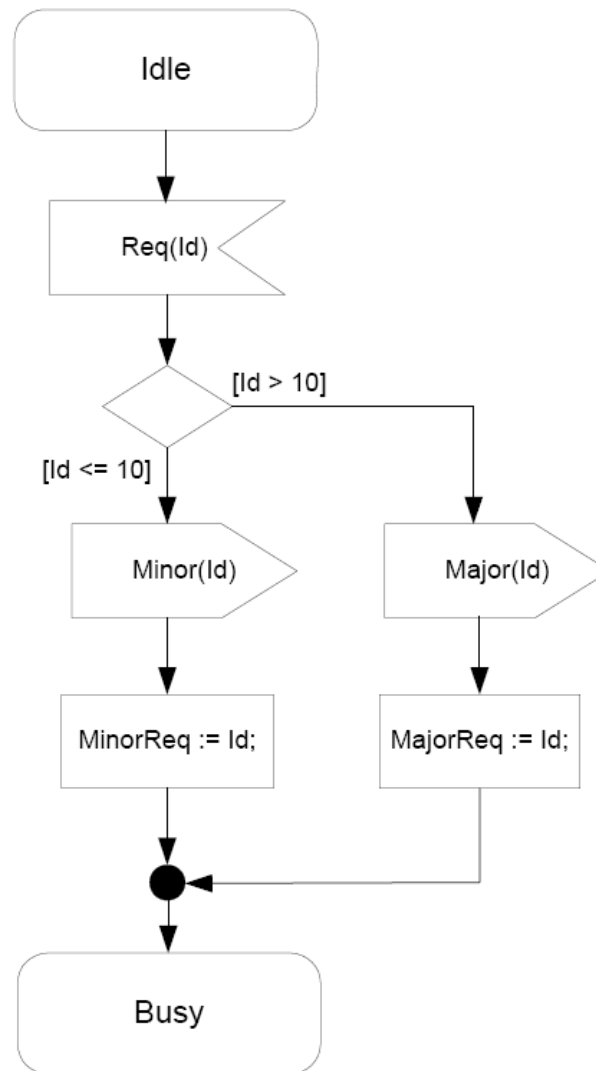
Initial

→ A pseudostate representing a default vertex that is the source for a single transition to the default state of a region.

❑ There can be at most one initial vertex in a region.

❑ Can have at most one outgoing transition.

❑ The outgoing transition from the initial vertex may have a behavior, but not a trigger or guard.

Final State

→ A special kind of state signifying that the enclosing region is completed.

❏ If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, the entire state machine is completed.

Shallow History

→ A pseudostate representing the most recent active substate of its containing state, but not the substates of that substate.

❏ A composite state can have at most one shallow history vertex.

❏ A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state.

❏ At most one transition may originate from the history connector to the *default shallow history state*. This transition is taken in case the composite state had never been active before.
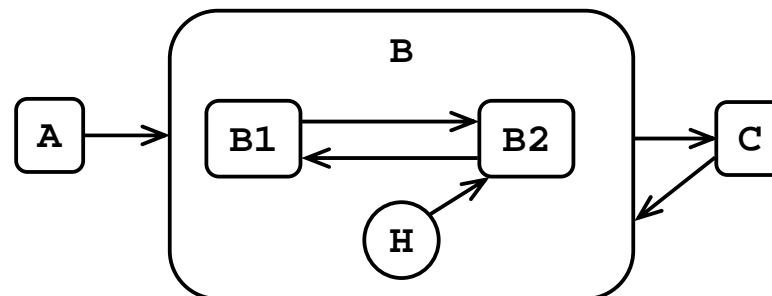
Deep History

→ A pseudostate representing the most recent active configuration (including recursive substates) of the composite state that directly contains this pseudostate, i.e., the state configuration that was active when the composite state was last exited.

❏ A composite state can have at most one deep history vertex.

❏ At most one transition may originate from the history connector to the *default deep history state*. This transition is taken in case the composite state had never been active before.
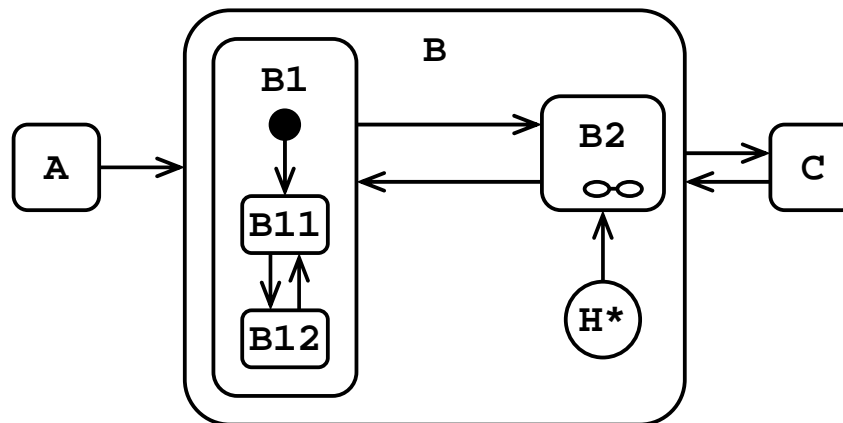
Entry and Exit Points

Entry Point

→   A pseudostate which is an entry point of a state machine or a composite state.

❑   In each region of the state machine or composite state it has a single transition to a vertex within the same region.

Exit Point

→   A pseudostate which is an exit point of a state machine or composite state.

❑   Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state.

Fork and Join
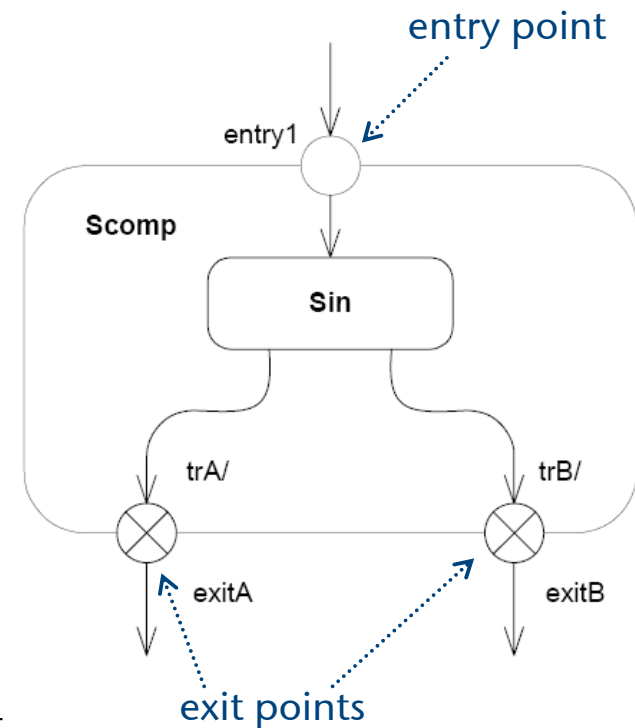
Fork

→ A pseudostate used to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state).

❒ The segments outgoing from a fork vertex must not have guards or triggers.

❒ Must have exactly one incoming and at least two outgoing transitions.

Join

→ A pseudostate used to merge several transitions emanating from source vertices in different orthogonal regions.

❒ The transitions entering a join vertex cannot have guards or triggers.

❒ Must have at least two incoming transitions and exactly one outgoing transition.
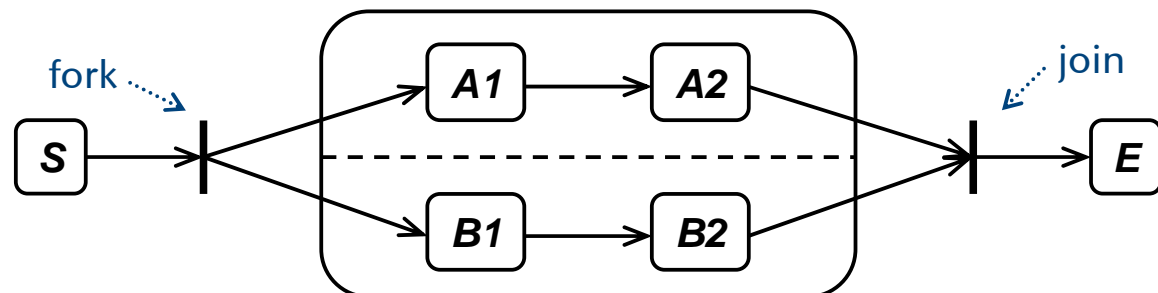
Junction

→ A pseudostate used to chain together multiple transitions.

❑ No semantics impact, can be replaced by multiple transitions.

❑ Constructs compound transition paths between states used to merge and/or split transitions.

❑ Must have at least one incoming and one outgoing transition.
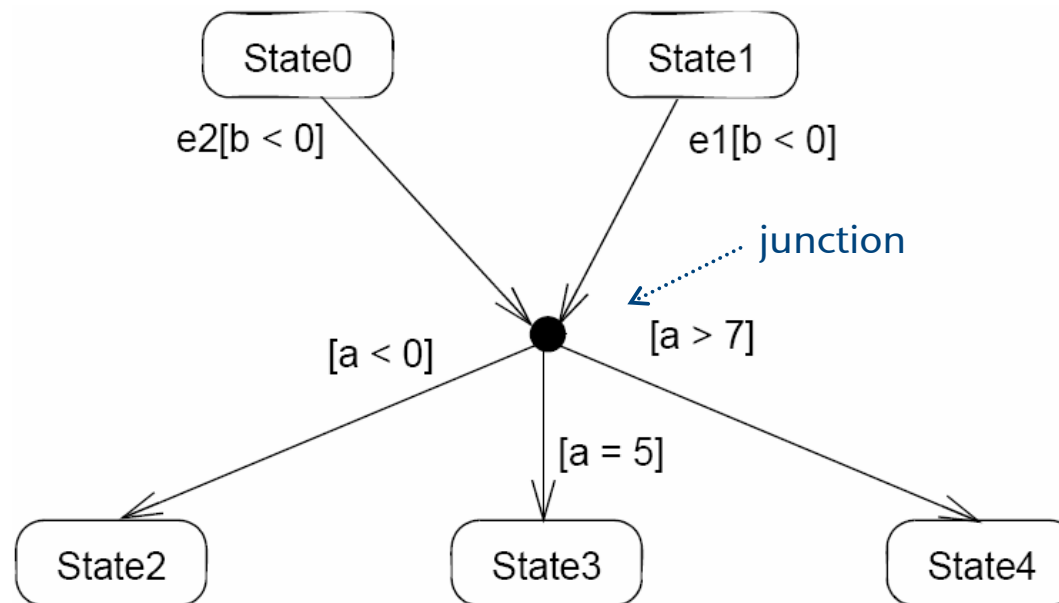
Choice

→ A pseudostate which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a *dynamic conditional branch*.

❑ Allows splitting of transitions into multiple outgoing paths such that the decision on which path to take.

❑ The decision may be a function of the results of prior actions performed in the same execution step.

❑ If more than one of the guards evaluates to true, an arbitrary one is selected.

❑ If none of the guards evaluates to true, the model is considered ill-formed.

❑ Predefined "else" guard can be used for one outgoing transition.

❑ Must have at least one incoming and one outgoing transition.
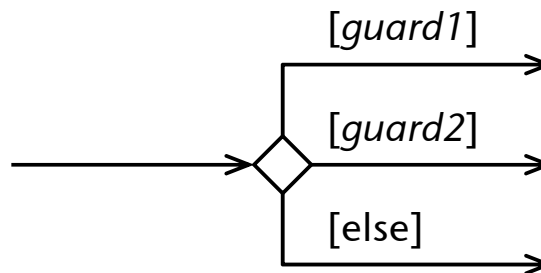
Terminate

→ A pseudostate used to terminate the execution of its state machine by means of destroying its context object.

❐ The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate.

✕

Protocol State Machine

→ A variant of state machine that specifies which operations of the *context classifier* can be called in which state and under which condition, i.e., determines the allowed call sequences on the classifier's operations.

❑ Presents the possible and permitted transitions on the instances of its context classifier, together with the operations that carry the transitions.

  ● In this way an instance usage *lifecycle* can be created.

❑ All transitions of a protocol state machine must be protocol transitions.

❑ Protocol state machines cannot have deep or shallow history pseudostates.

```
stm Name {protocol}
```
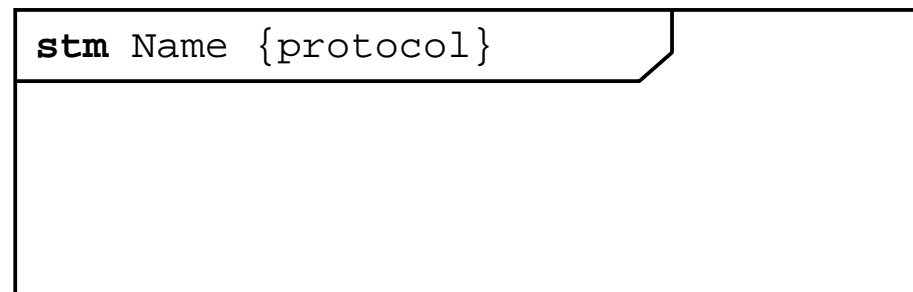
Example of Protocol State Machine
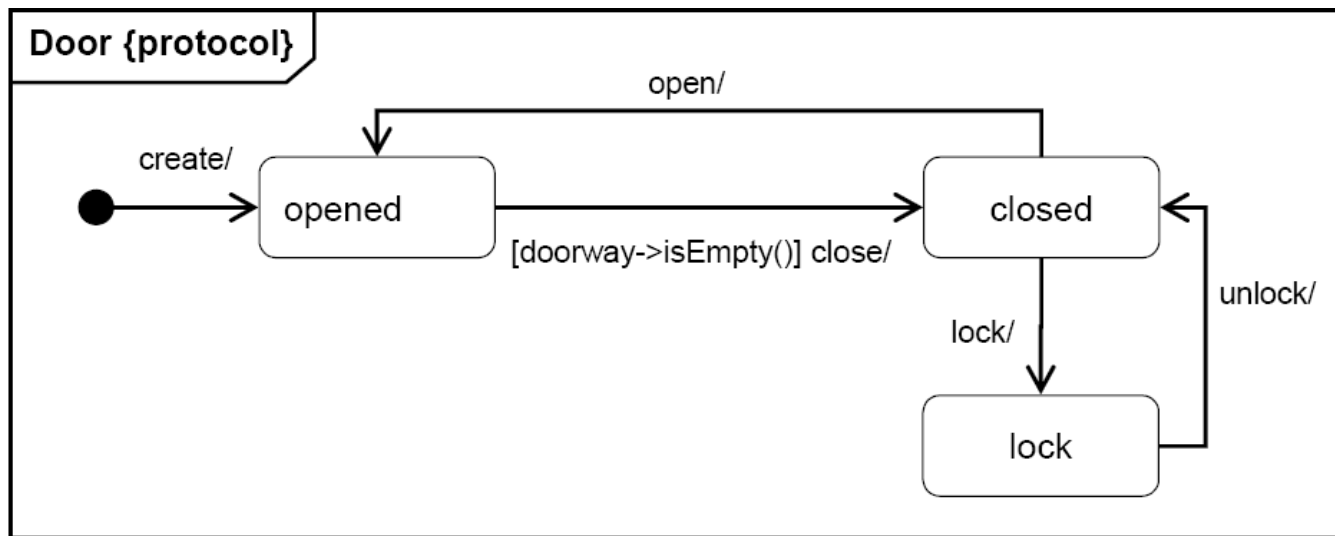
State in Protocol State Machines

❑ An exposed stable situation of its context classifier.

❑ The state represents the situation when an instance of the classifier is not processing any operation.

❑ Can specify an *invariant*–condition(s) that is(are) always true when an instance of the classifier is in the current state.

- State invariants are additional conditions to the preconditions of the outgoing transitions, and to the postcondition of the incoming transitions.

❑ Cannot have entry, exit, or do internal behaviors.

```
   Name
[invariant]
```

Protocol Transition
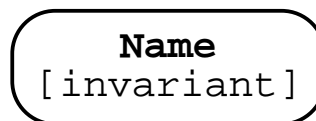
→ A specialized transition that specifies a legal transition for an operation.

❏ Specifies:

- *Pre-condition*: the condition that must be true at triggering the transition.

- *Operation*: a reference to the context's operation which represents its call at the given position in the protocol state machine.

- *Post-condition*: the condition that should be obtained once the transition is accomplished.

❏ No behavior (action) is specified for protocol transition.

$$[precondition]\ operation\ /\ [postcondition]$$

Overall Semantics of State Machines

☐ Event occurrences go into the (FIFO) *event pool*.

☐ Event occurrences are processed one at a time, but only if processing of the previous event completed; except of *do activities* of states (see later). This is called **run-to-completion** mechanism.

☐ One event occurrence may result in one or more transitions being enabled for firing (each in one orthogonal region).

- All enabled (non-conflicting) transitions are fired.

- The order in which selected transitions fire is not defined.

- When all orthogonal regions have finished executing the transition, the current event occurrence is fully consumed.

☐ If no transition is enabled, the event occurrence is discarded.

☐ During a transition, a number of actions may be executed. If such an action is a synchronous operation invocation on an object executing a state machine, then the transition step is not completed until the invoked object completes its action.

Overall Semantics of State Machines (cont.)

❑ In case of conflicting transitions (for instance, two transitions with the same event and the guards both satisfied, originating from the same state), only one of them will fire.
If *t1* is a transition whose source state is *s1*, and *t2* has source *s2*, then:

- If *s1* is a direct or transitively nested substate of *s2*, then *t1* has higher priority than *t2* ⇒ *t1* is fired.

- If *s1* and *s2* are not in the same *state configuration* (an execution state of the state machine), then there is no priority difference between *t1* and *t2*.

Process of State Machine Modeling

1. Identify possible states.

2. Identify external events which can cause changes of states.

3. To each state attach transitions based on relevant events.

4. Add internal actions and transitions.

5. Identify composite states and specify contents of regions.

6. Repeat until the state machine is complete.

7. Review the state machine by simulation of the owner's lifecycle. Based on simulated events, check the appropriate triggering of transitions and execution of behavior.

Object-Oriented Analysis and Modeling

UML Auxiliary Constructs

Auxiliary Constructs

→ **Various auxiliary modeling mechanisms.**

Comprise:

- Information flows.

- Models.

- Primitive types.

- Templates.

Information Flows

□ Provides mechanisms for specifying the exchange of information between entities of a system at a high level of abstraction.

□ Do not specify the nature of the information (type, initial value), nor the mechanisms by which this information is conveyed (message passing, signal, common data store, parameter of operation, etc.), nor sequences or any control conditions.
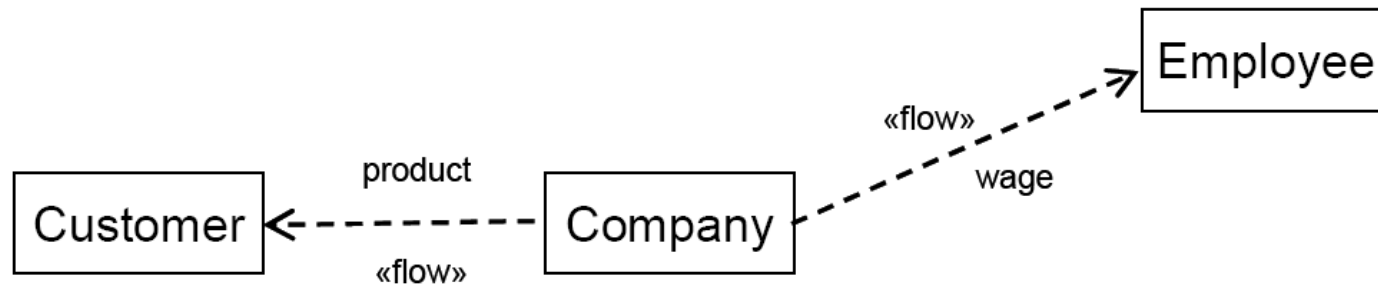
Information Flow

→ Specifies that one or more information items circulates from its sources to its targets.

❑ It is used to **abstract** the communication.

❑ Requires some kind of "information channel" for transmitting information items.

- Connectors, links, associations, or even dependencies.

❑ When a source or a target is:

- A *classifier*, the information flow represents *all the potential instances* of the classifier.

- A *part*, the information flow represents *all instances* that can play the role specified by the part.

- A *package*, the information flow represents all potential *instances* of the directly or indirectly *owned classifiers of the package*.

```
                    information item
┌────────┐       - - - - - - - - - - - ->┌────────┐
│ Source │                               │ Target │
└────────┘           «flow»              └────────┘
```
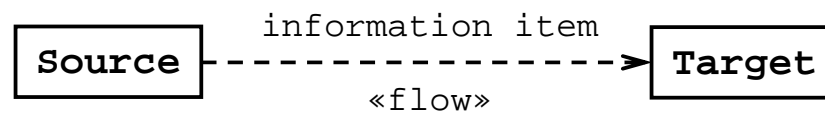
Information Item

→ An abstraction of all kinds of information that can be exchanged between objects.

❑ A kind of *classifier* intended for representing information at a very abstract way, one which *cannot be instantiated*.

❑ Used to define preliminary models, before having made detailed modeling decisions on types or structures.

❑ Encompasses all sorts of data, events, facts that are exploited inside the modeled system.

❑ An information item does not specify the structure (properties or associations), the type, or the nature of the represented information.

❑ Information items can be decomposed into more specific information items using *representation links* between them.

```
<<information>>
    Name
```

```
              ▶
    Name
```
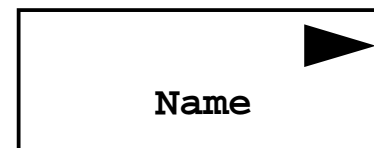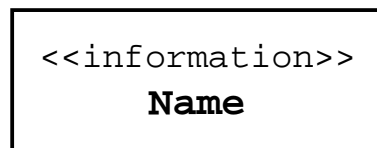
Examples of Information Flows

Definition of information items:
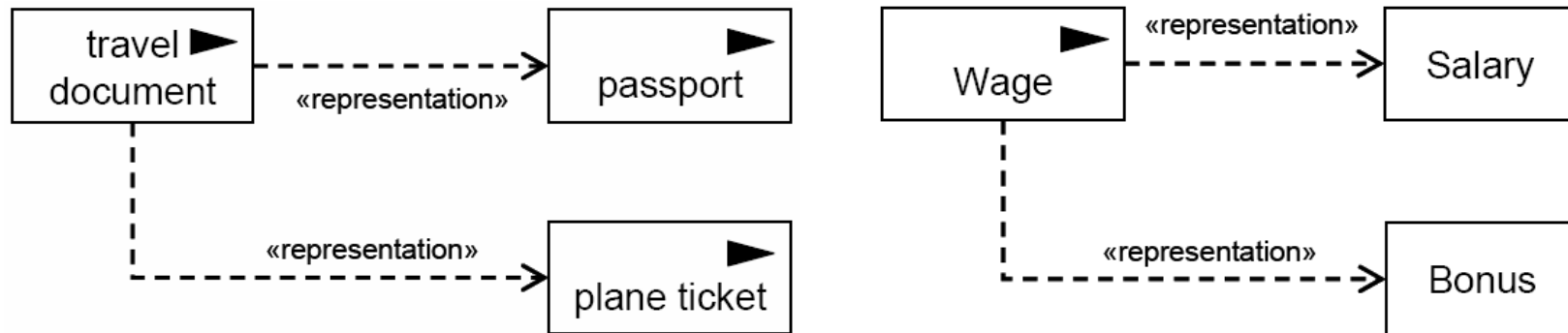
Information flows attached to a connector:
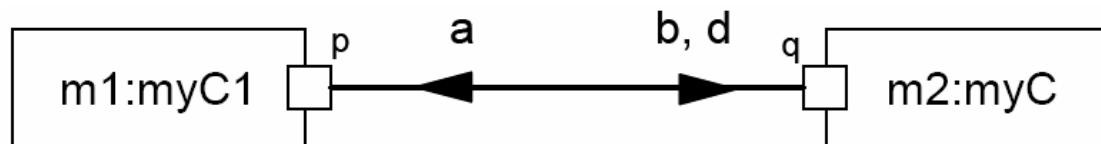
**Information flows attached to an association:**
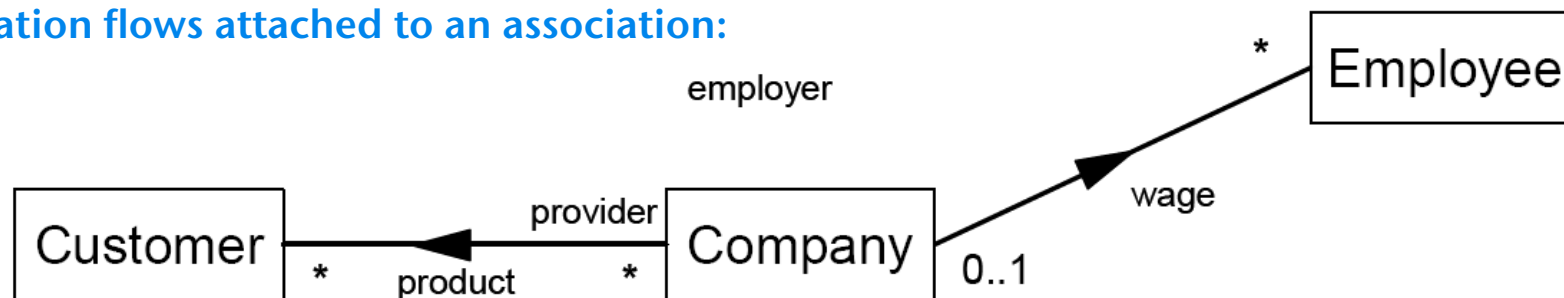
# Models

❒ Allows to explicitly represent models.
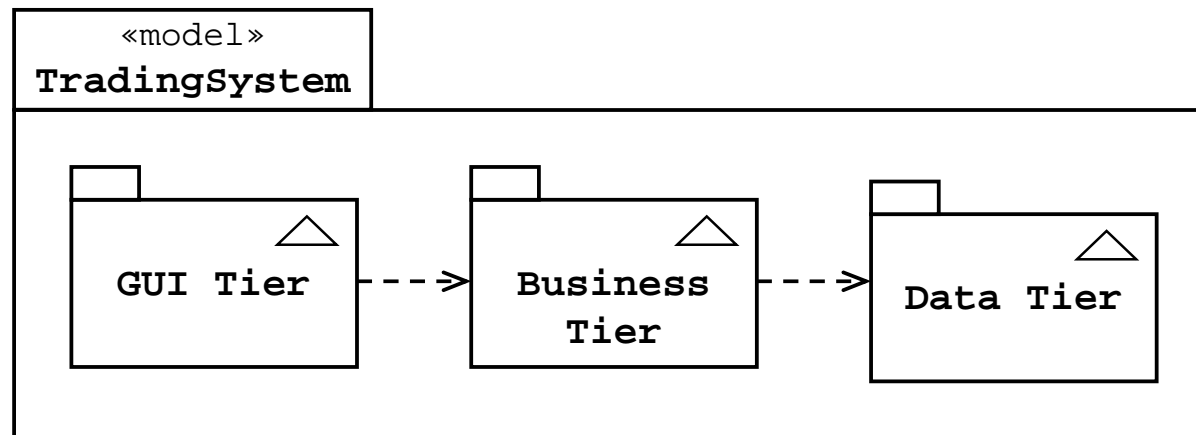
Model

→ Used to capture a view of a physical system.

❑ It is an *abstraction* of the physical system, with a certain *purpose*.

- This purpose determines what is to be included in the model and what is irrelevant. It also determines the readers of the model.

- Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

❑ Examples of usages: logical model, behavioral model, deployment model, …

❑ Models are "self-contained".

❑ A specialized package.

```
<<model>>
   Name
```

```
        △
   Name
```
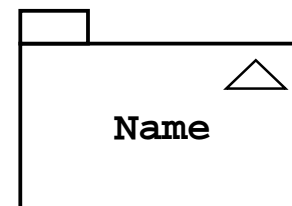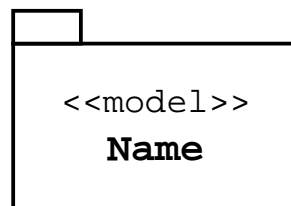
Primitive Types

- A set of primitive data types used in the UML metamodel.

- These types are reused by both MOF and UML, and may potentially be reused also in user models.

- Tool vendors, however, typically provide their own libraries of data types to be used when modeling with UML.

«primitive» **Integer**

«primitive» **Boolean**

«primitive» **String**

«primitive» **UnlimitedNatural**
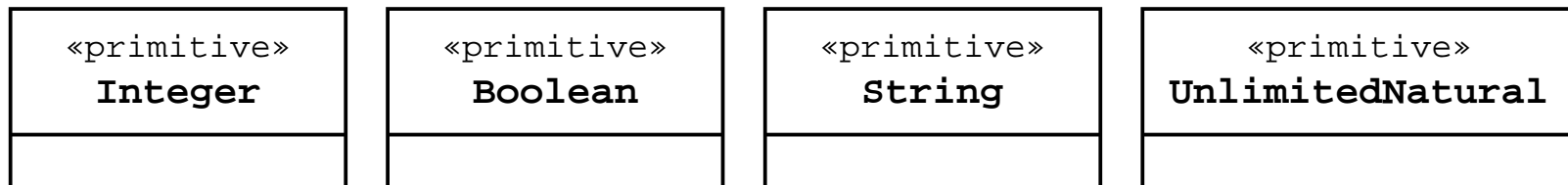
Primitive Types (cont.)

❑ *Integer*

- A primitive type representing integer values.

❑ *Boolean*

- Used for logical expressions. It has the predefined literals *true* and *false.*

❑ *String*

- A sequence of characters in some character set. Defines a piece of text.

❑ *Unlimited natural*

- An instance of unlimited natural is an element in the (infinite) set of naturals (0, 1, 2...). The value of infinity is shown using an asterisk ('*').

- Appears as the type of upper bounds of multiplicities in the metamodel.

Templates

□ The mechanisms for defining reusable templates and bound elements in general, and the specialization of these for classifiers and packages.

Template Definition
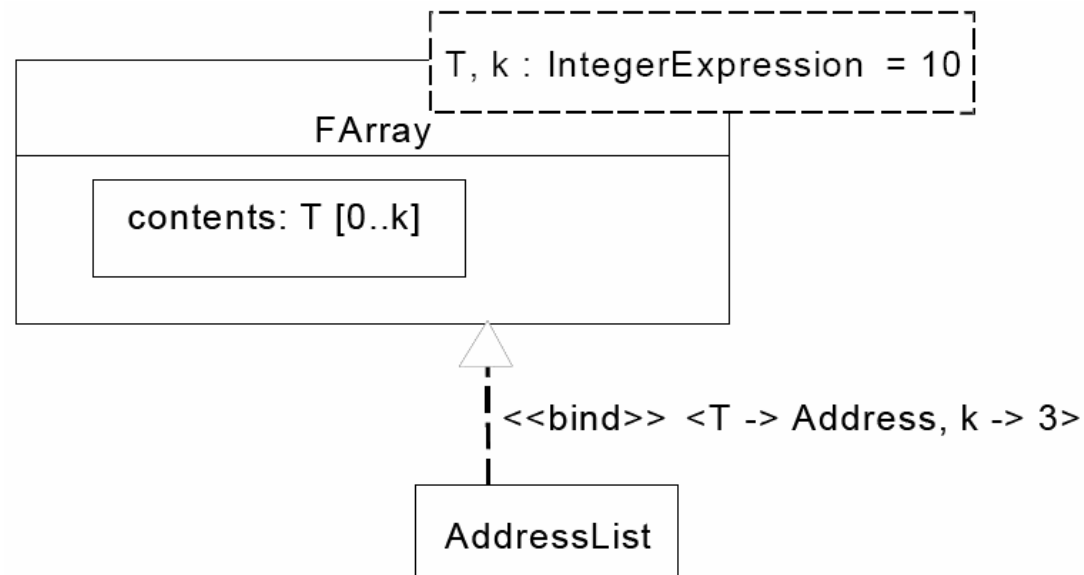
→ Definition of a template signature for a templateable elemement.

❑ *Templateable element*

- An element which can contain a template signature.

- A templateable element that contains a template signature is often referred to as a **template**.

❑ *Template signature*

- Specifies a set of formal template parameters for a templated element.

- Is owned by a templateable element and has one or more template parameters.

❑ *Template parameter*

- References a parameterable element that is exposed as a formal template parameter in the containing template.

- Format:

  template-parameter ::= template-param-name [':' *parameter-kind*] ['=' *default*]

Template Definition (cont.)

- ❏ *Parameterable element*
  - An element that can be exposed as a formal template parameter for a template, or specified as an actual parameter in a binding of a template.

- ❏ *Named element (from Templates)*
  - A named element is extended to support using a *string expression* to specify its name.
  - Used to allow names of model elements to involve template parameters.
  - When a template is bound, the sub expressions are substituted with the actual values substituted for the template parameters to obtain the actual bound element name.
  - Format:
    - The string expression appears between "$" signs.
    - Template parameters in are enclosed in '<' and '>' signs.
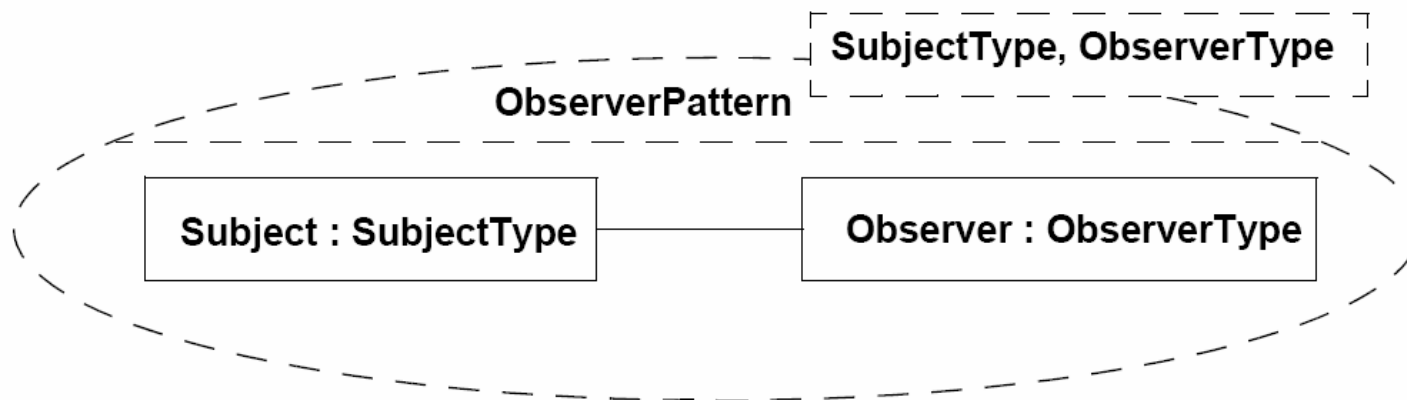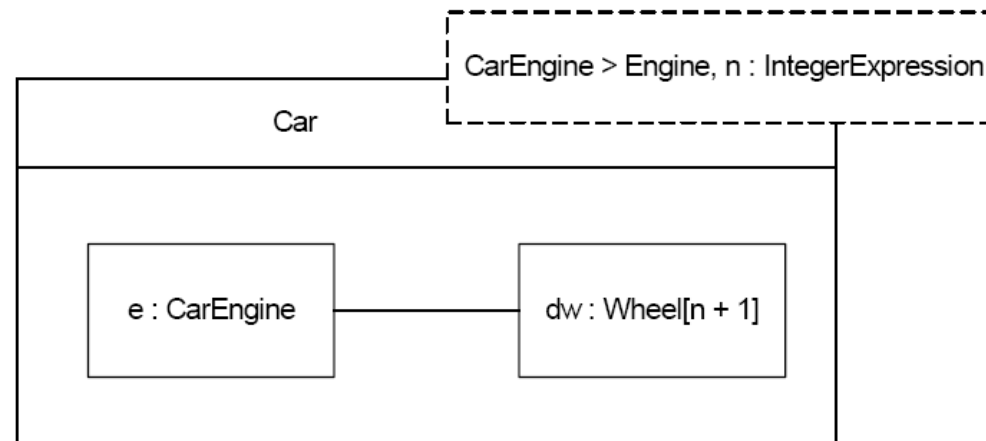
Template Binding

→ Represents a relationship between a templateable element and a template. A template binding specifies the **substitutions** of actual parameters for the formal parameters of the template.

❐ Format :

element-name ':' binding-expression [',' binding-expression]*

binding-expression ::= template-element-name '<'
*   template-parameter-substitution [',' template-parameter-substitution]* '>'*

template-param-substitition ::= template-param-name '->' actual-template-
*   parameter*

```
┌─────────────┐
│  Template   │
└─────────────┘
       ▲
       ┊    «bind» <substitition>
┌─────────────┐
│ BoundElement│
└─────────────┘
```

```
┌────────────────────────┐
│ BoundElement:Template  │
│     <substitution>     │
└────────────────────────┘
```
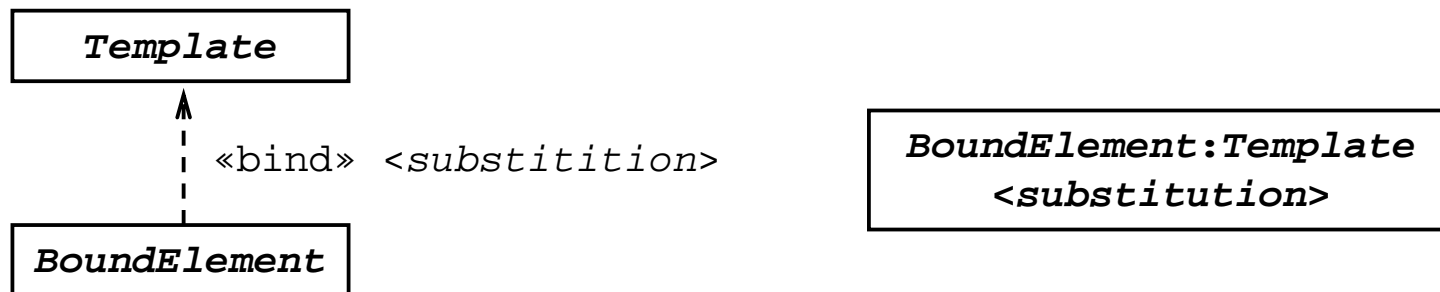
Examples of Template Bindings (1)

FArray <T -> Point>

DieselCar : Car<CarEngine -> DieselEngine, n -> 2>

Observer

ObserverPattern<SubjectType -> CallQueue, ObserverType -> SlidingBarIcon>

Examples of Template Bindings (2)
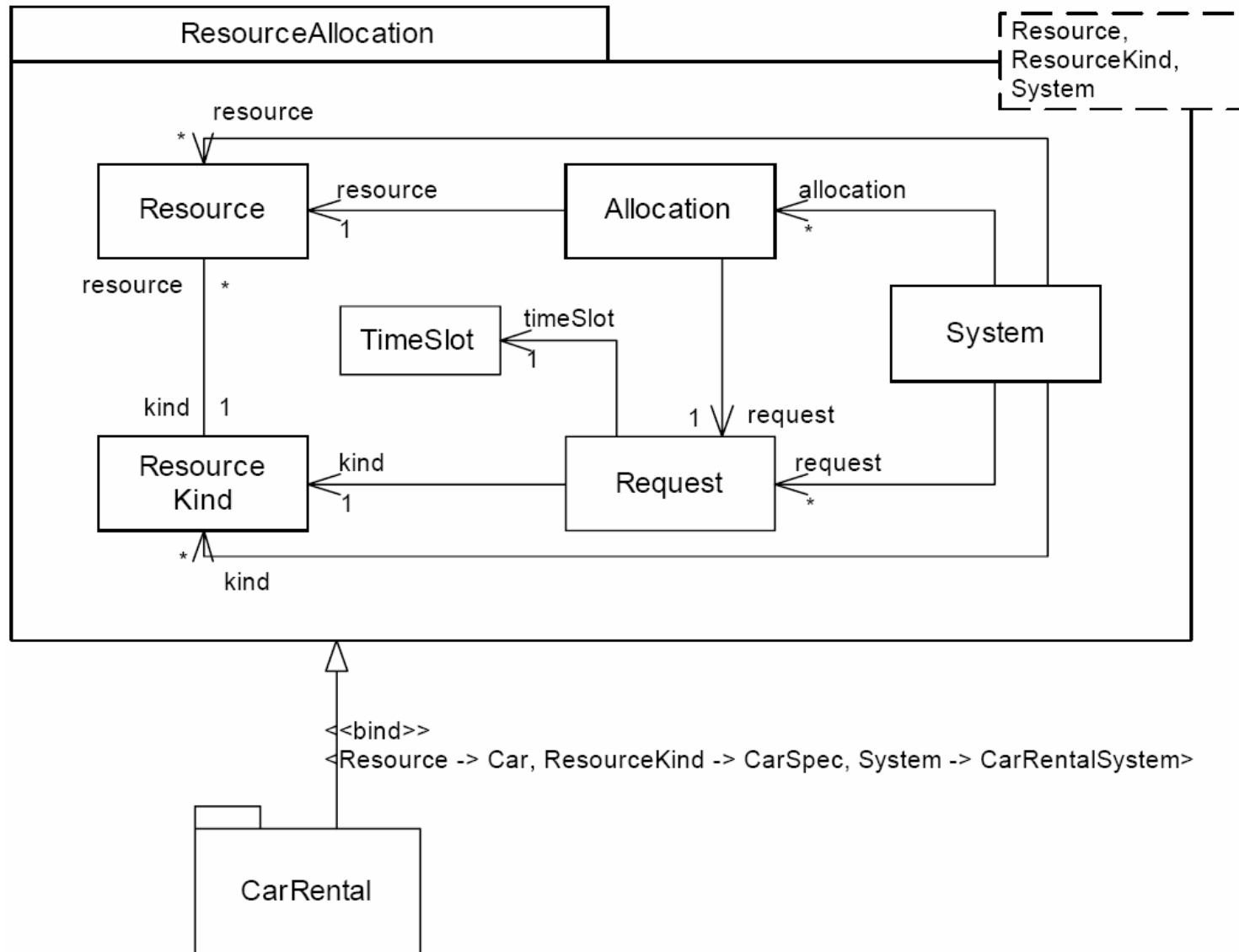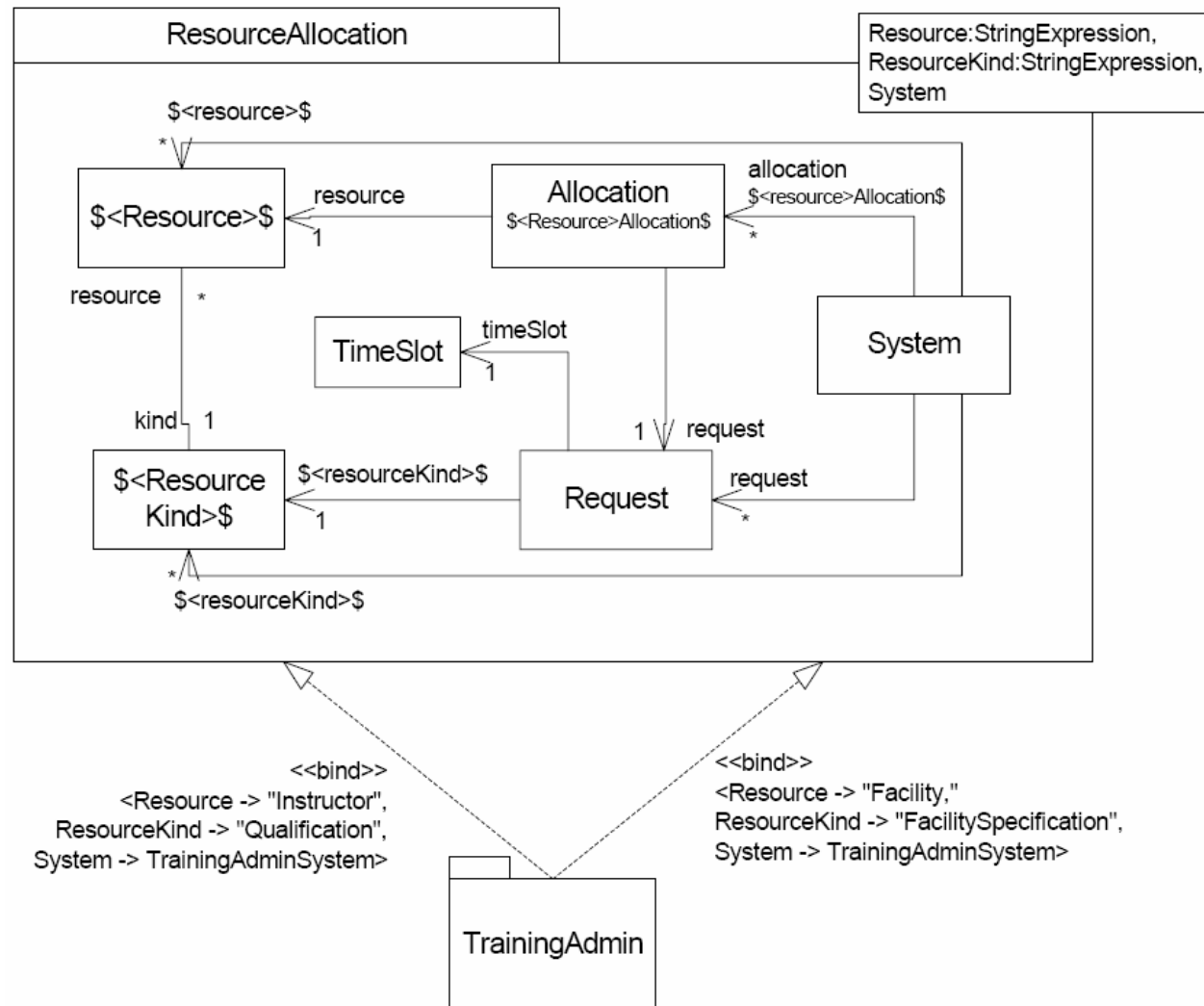
# Object-Oriented Analysis and Modeling

UML Profiles

UML Profiles

→ **The mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes, e.g., to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time systems or business process modeling).**

❒ It is a *conservative extension mechanism*, not a *first-class extension mechanism*, i.e., it does not allow for modifying existing metamodels.

❒ Sometimes also referred to as "UML extension mechanisms".

Profile

→ Kind of package that extends a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.

❑ The primary extension construct is the stereotype.

❑ A profile is a restricted form of a metamodel that must always be related to a reference metamodel, such as UML.

- A reference metamodel typically consists of metaclasses that are either imported or locally owned.

- All metaclasses that are extended by a profile have to be members of the same reference metamodel.

- Applying a profile does not change the underlying model in any way.

❑ By application a profile, the following elements are visible:

1. referenced by an explicit metaclass reference, or

2. contained (directly or transitively) in a package that is referenced by an explicit metamodel reference, or

3. extended by a stereotype owned by the applied profile.

- All other model elements are hidden (not visible) when the profile is applied.

Examples of Profiles
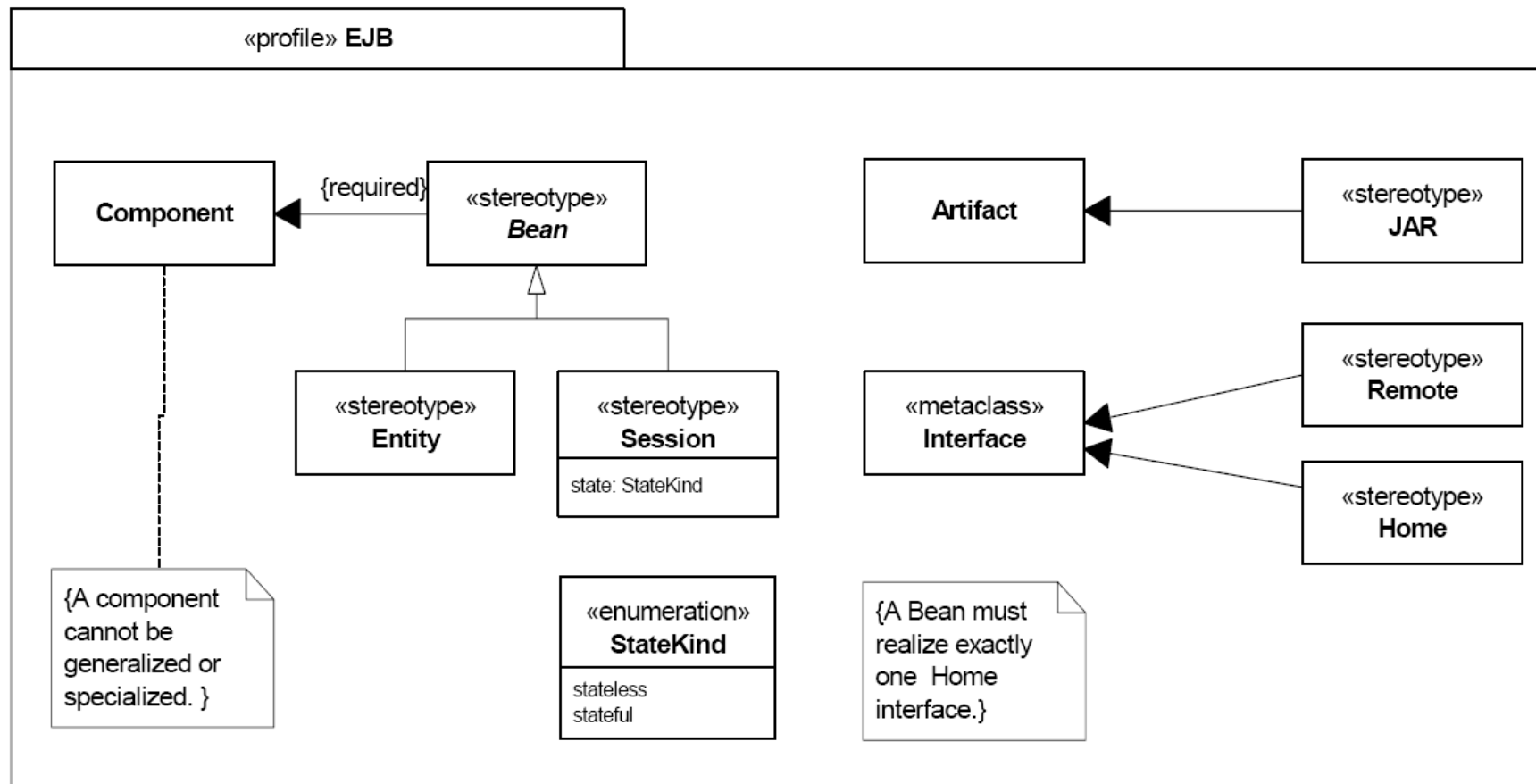
Metaclass Reference

→ References a metaclass that may be extended.
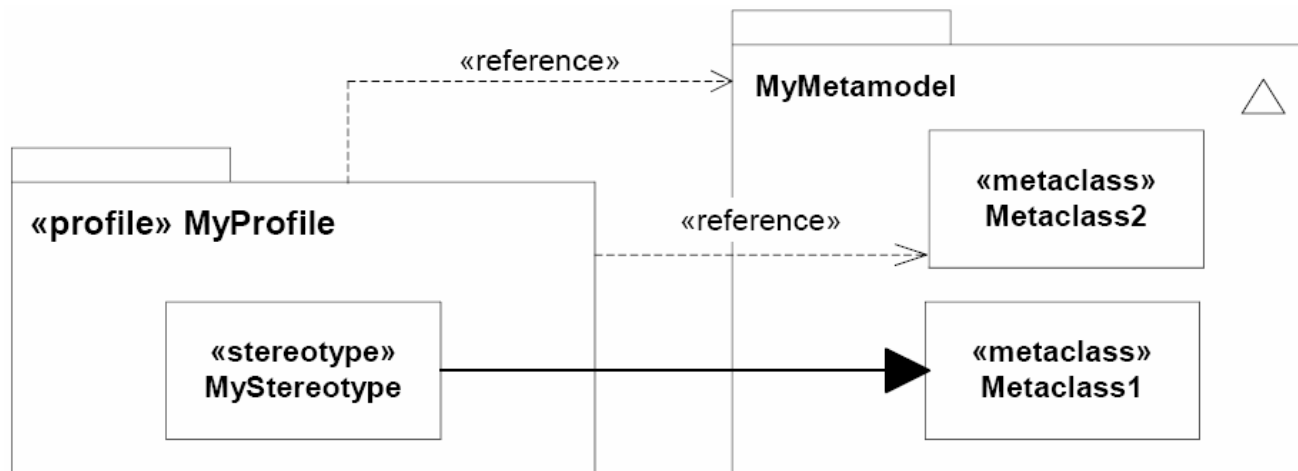
❏ A specific element import.
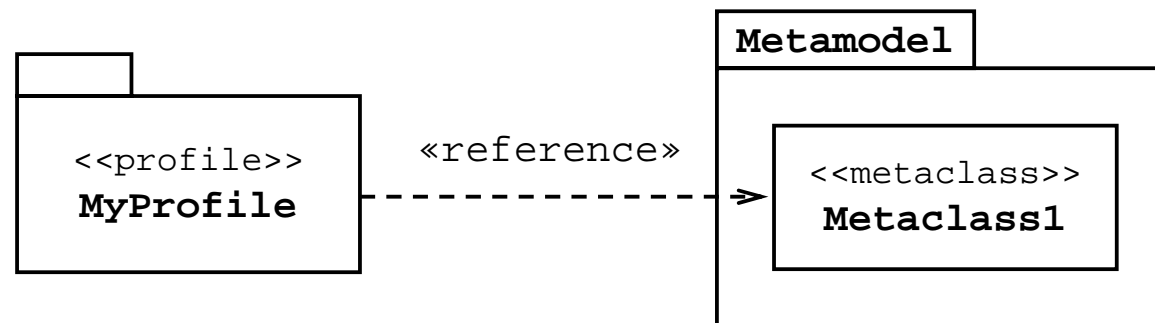
Metamodel Reference

→ References a package containing (directly or indirectly) metaclasses that may be extended.
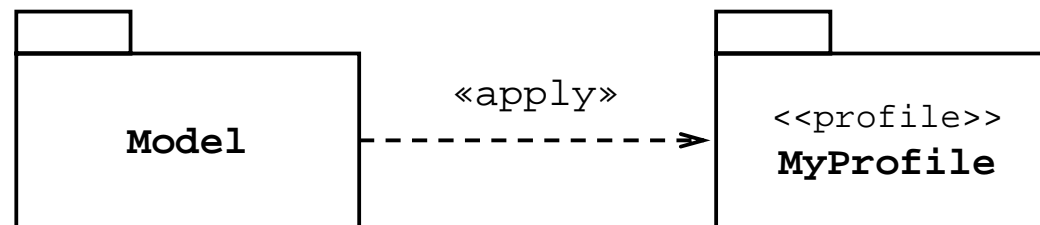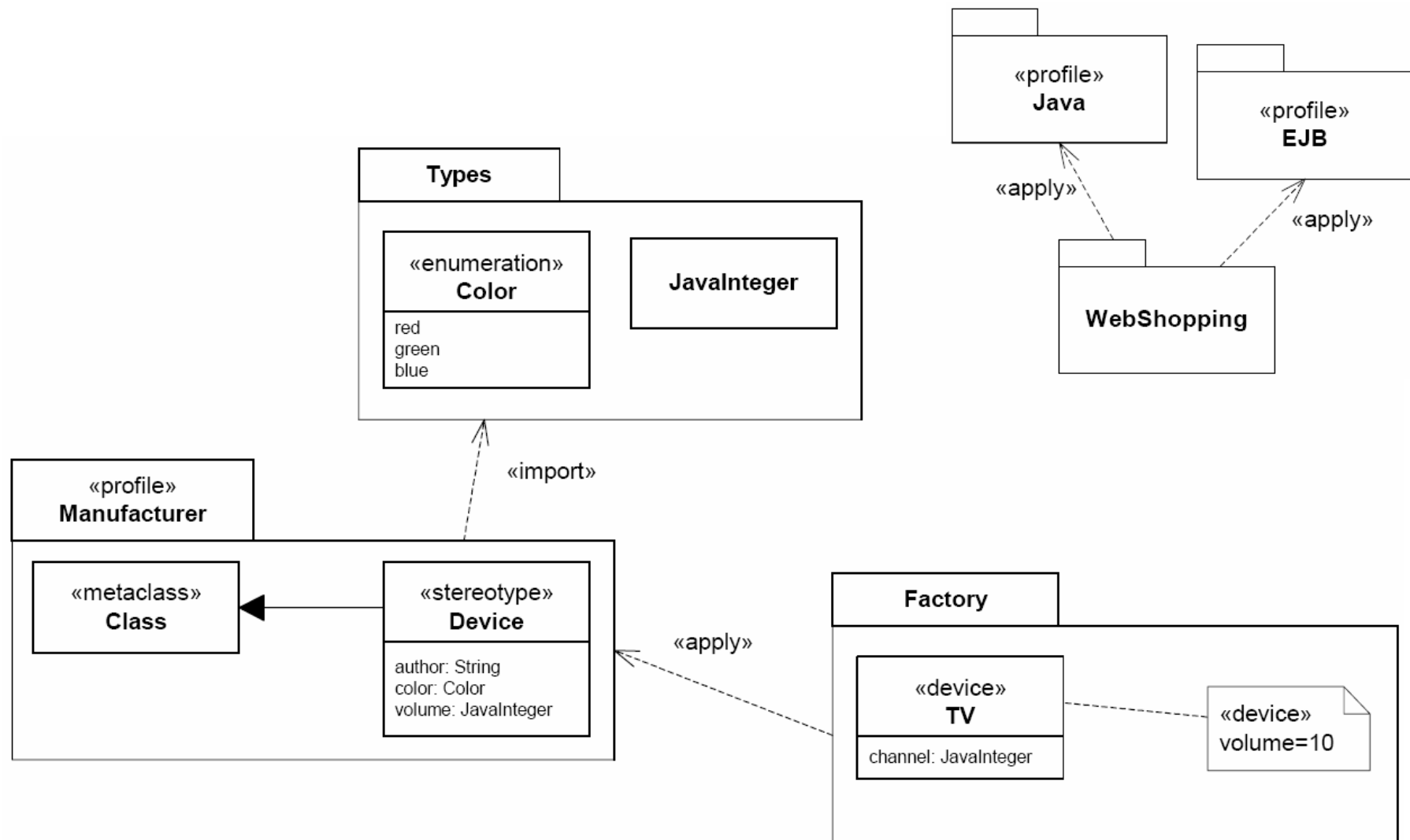
❏ A specific package import.
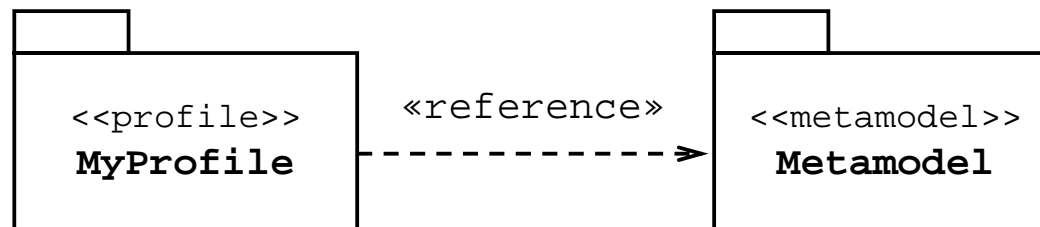
Example of References

Profile Application

→ A relationship used to show which profiles have been applied to a package.

❏ One or more profiles may be applied to a package.

- It is possible to apply multiple profiles to a package as long as they do not have conflicting constraints.

❏ Applying a profile means that it is allowed, but not necessarily required, to apply the stereotypes that are defined as part of the profile.

❏ If a profile that is being applied depends on other profiles, then those profiles must be applied first.

❏ When a profile is applied, instances of the appropriate stereotypes should be created for those elements that are instances of metaclasses with required extensions.

```
 _____                        _____
| |__|    |        «apply»      | |__|          |
|         | - - - - - - - - - ->|  <<profile>>  |
|  Model  |                     |  MyProfile    |
|_____|                     |_____|
```

Examples of Profile Applications
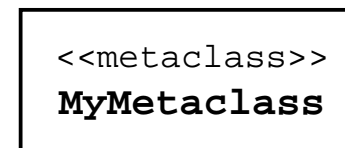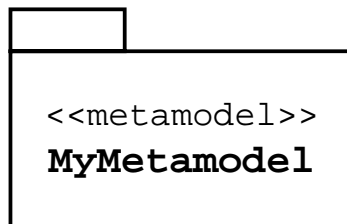
Metamodel and Metaclass

Metamodel

→ A model that defines the modeling concepts (their abstract syntax) of a modeling language.

❏ A stereotyped model.

Metaclass

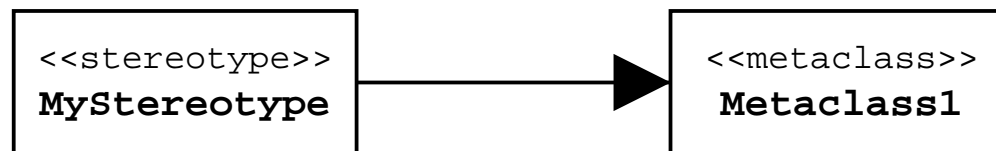→ A class whose instances are modeling elements or their parts.

❏ A specialized class.

```
┌───────────────────────────┐
│ <<metamodel>>             │
│ MyMetamodel              │
└───────────────────────────┘
```

```
┌───────────────────────────┐
│     <<metaclass>>         │
│     MyMetaclass          │
└───────────────────────────┘
```

Stereotype

→ Specification of how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass.

❏ May have properties used as *tag definitions*.

❏ When a stereotype is applied to a model element, the values of the properties may be referred to as *tagged values*.

❏ Must extend one, or more, metaclasses.

- Any model element from the reference metamodel can be extended by a stereotype.
- For example in UML, states, transitions, activities, use cases, components, attributes, dependencies, etc. can all be extended.

❏ Can define one or more images.

❏ A specialized Class.

`<<stereotype>>` **MyStereotype**
`property1` `property2`

Extension
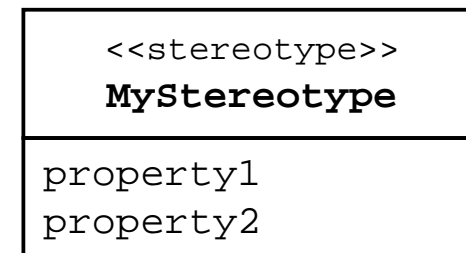
→ Is used to indicate that the properties of a metaclass are extended through a stereotype.

❑ {required} is used to indicate whether an instance of the extending stereotype must be created when an instance of the extended metaclass is created.

❑ A specialized association.

❑ Note: Stereotype (from UML) is the only kind of metaclass that cannot be extended by stereotypes.

```
+----------------------+        +----------------------+
| <<stereotype>>       |------->| <<metaclass>>        |
| MyStereotype         |        | Metaclass1           |
+----------------------+        +----------------------+
```

Examples of Extensions

Copyright © 2007 by Whitestein Technologies, Switzerland
All rights reserved.
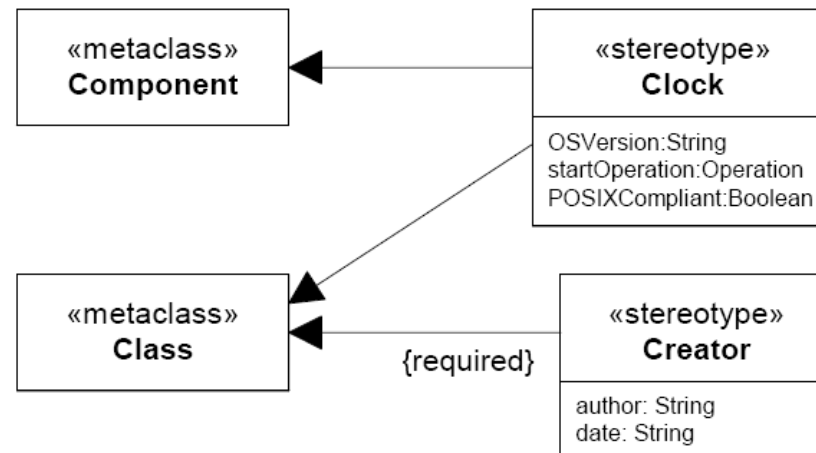
Application of a Stereotype

- ❑ Application of a stereotype in a modeling element.

- ❑ Format:

  '<<' stereotype-name '>>' ['<<' stereotype-name '>>']*

  - Placed above or before the name of the model element.
  - Alternatively (also), an icon (or multiple icons) can be used.

- ❑ Format of tagged values:

  namestring '=' valuestring

  valuestring ::= value [',' value]*

  - In a comment connected to the stereotyped element.
  - In a separate compartment.
  - Above the name strng.
  - If the type of property is Boolean and a value is omitted, the value is implicitly true.

Examples of Stereotype Applications

Copyright © 2007 by Whitestein Technologies, Switzerland
All rights reserved.
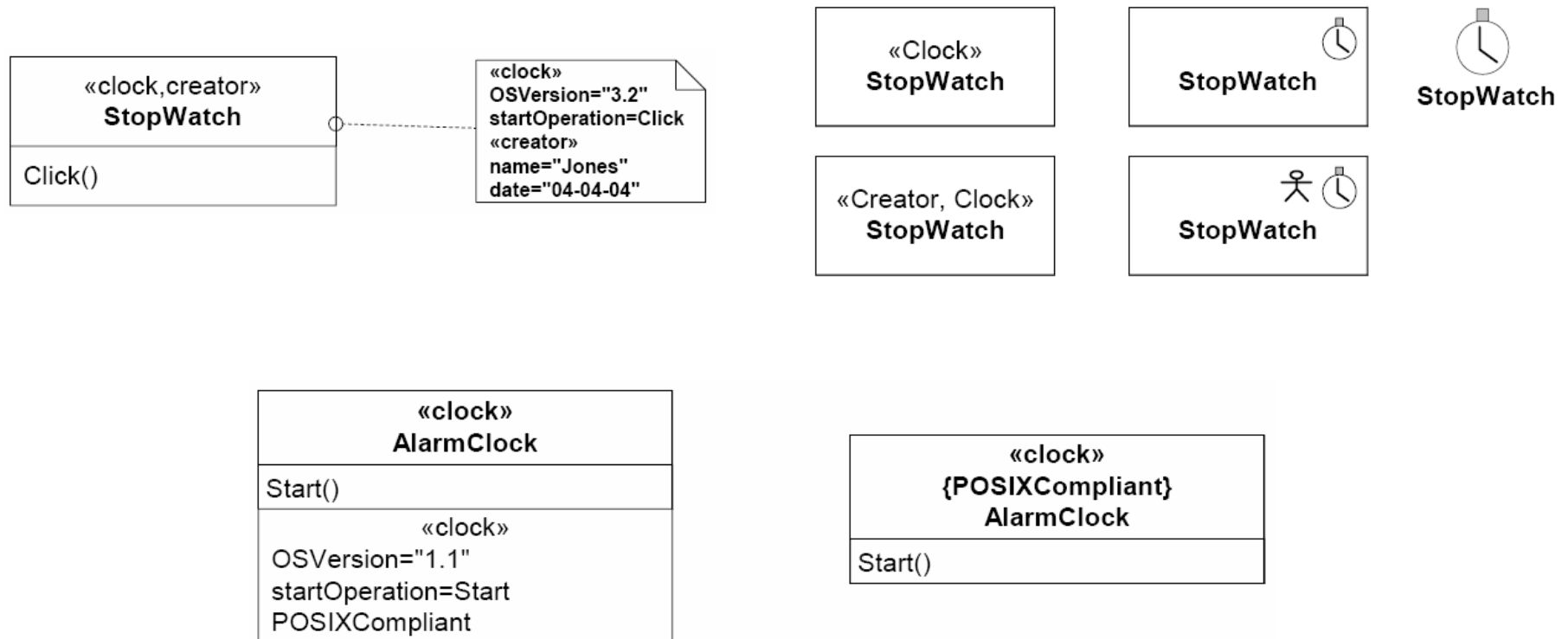
Object-Oriented Analysis and Modeling

Analysis Data Patterns

Analysis Data Patterns

→ **Analysis data patterns describe the common problems in modeling data and provide standard generic and repeatable solutions.**

❑ Purpose:

- To document good practices in data modeling.

- To teach (junior) analysts how to build better models.

- To ease modeling of standard situations and improve readability of models.

❑ Modeling notation: a simplified version of class diagrams.

- Classes, associations, generalizations.

❑ Usually, they model the conceptual/domain level.

❑ Mostly oriented to the business area.

❑ There exist also other than analysis data patterns, e.g., *behavioral patterns* modeled as interactions, activities or state machines.

Standardization of Analysis Patterns

❑ The is no industrial (or other) standard of analysis patterns.

- In contrast to "GoF" *design patterns*, from Gamma, et. al.

❑ Several approaches (more-or-less incompatible) exist, e.g.:

- M. Fowler: Analysis Patterns: Reusable Object Models. Addison-Wesley Professional, October 1996.

- D. C. Hay: Data Model Patterns: Conventions of Thought. Dorset House Publishing Company, Inc., November 1995.

- (our main source) L. Sesera, A. Micovsky and J. Cerven: Architektura softverových systemov. Analyticke datove vzory. Slovenska technicka univerzita v Bratislave, 2000.

- ...

Types of Analysis Data Patterns

❏ Optimization "meta" patterns.

    → Domain-independent analysis patterns used to optimize data models in general.

❏ Business-related analysis data patterns.

    → Domain-specific data patterns.

    ● Usually explain a step-by-step improvement of a non-optimal model by several applications of "meta" patterns.

    ● Covered topics:

        – *People*: humans in a business who perform activities and use objects, classification of people, and their organizational structures.

        – *Objects*: artifacts used in a business, e.g., products, utilized material resources, tools, means, etc.

        – *Activities*: activities of a business process, and the related data, e.g., classification and relationships of activities, work orders, resource utilization, roles in activity execution, execution plans, etc.

        – *Trading*: buying and selling products, procurement of material and tools, orders, contracts, invoices, financial transactions, etc.

Optimization "Meta" Patterns

Abstraction of Types

- Original model—several entities that specify objects of different types:

Entity1	Entity2	Entity3

...
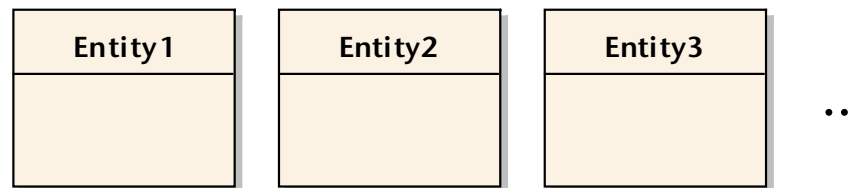
- Typing abstraction—different entities (*Entity1*, *Entity2*, ...) are represented by one (*Entity*) and its type is defined dynamically (in runtime):

```
┌─────────────┐
│ Entity Type │
├─────────────┤
│             │
│             │
└─────────────┘
       │ 1
       │
       │ *
┌─────────────┐
│   Entity    │
├─────────────┤
│             │
│             │
└─────────────┘
```
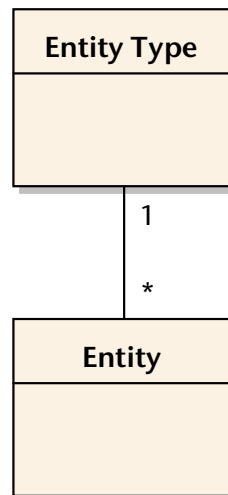
Abstraction of Relationships (Associations)

❏ Original model–entities with several relationships:
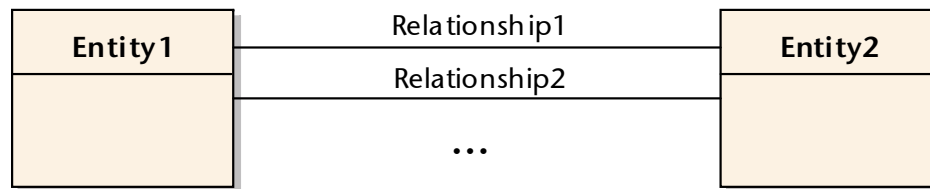
❏ Representing relationships as entities–used to dynamically manage entity relationships and their types:
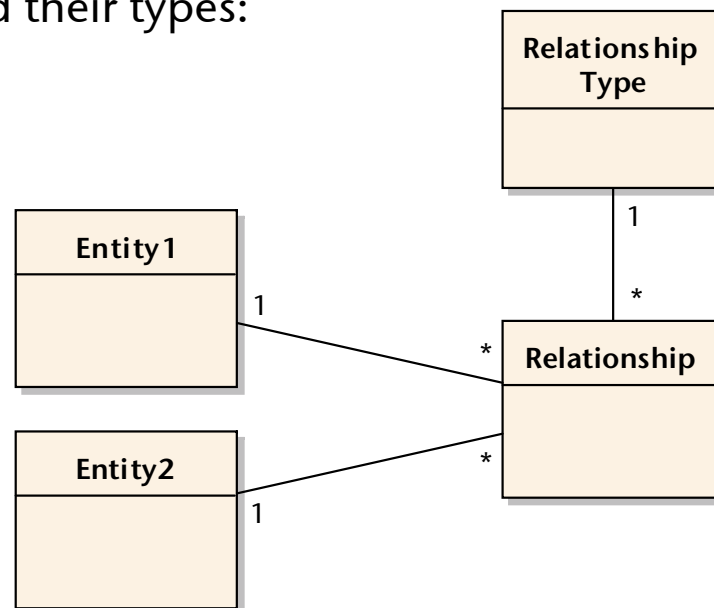
Optimization "Meta" Patterns (cont.)

◻ Simplified form of the relationship abstraction, where the related entities
have a common super-type:
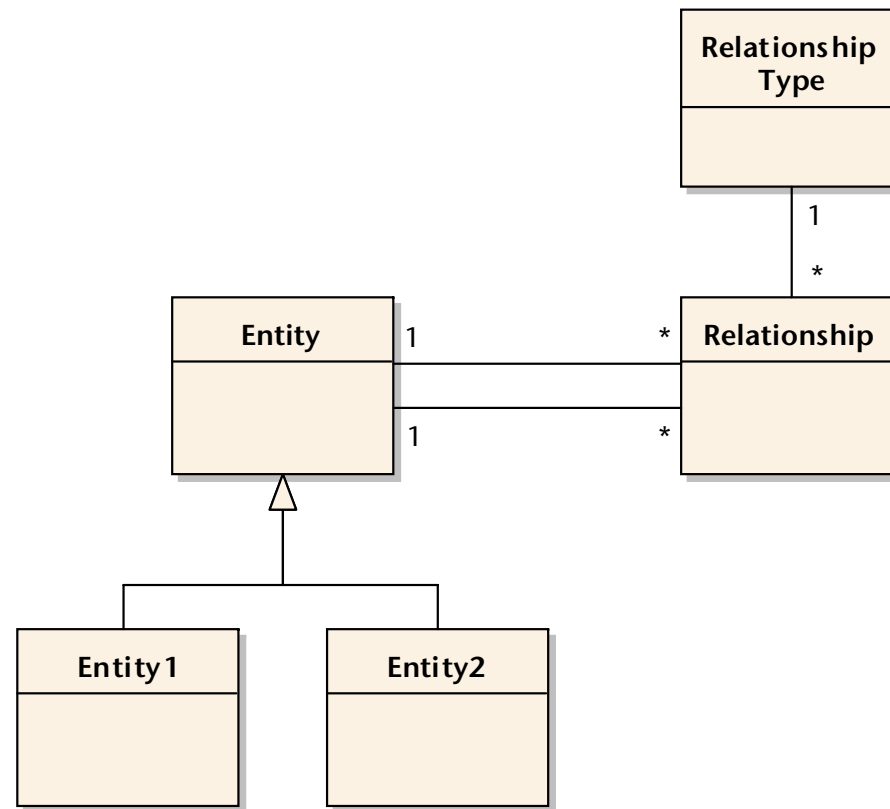
Abstraction of Attributes

❐ Original model–static set of attributes:

Entity
attribute1
attribute2
...
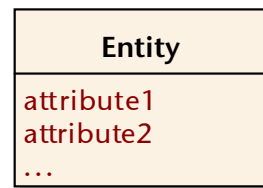
❐ Representing attributes as entities–dynamic set of attributes:
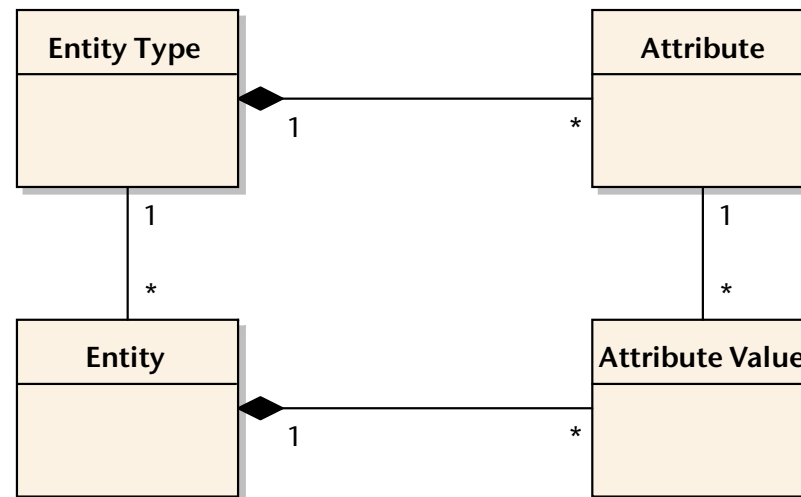
Example of Application "Meta" Patterns

☐ People in family, their attributes, and relationships.

Business Patterns–People
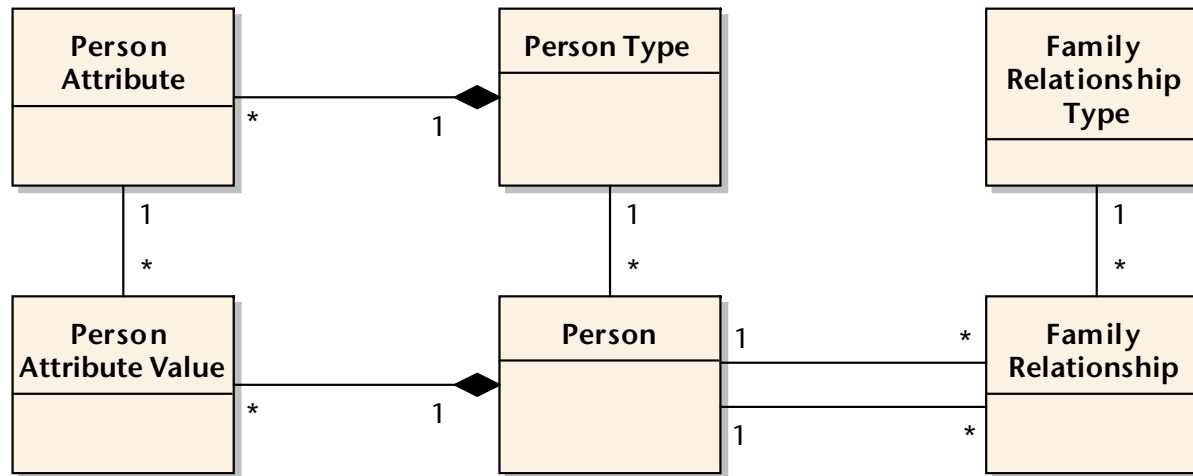
☐ Direct representation of positions:
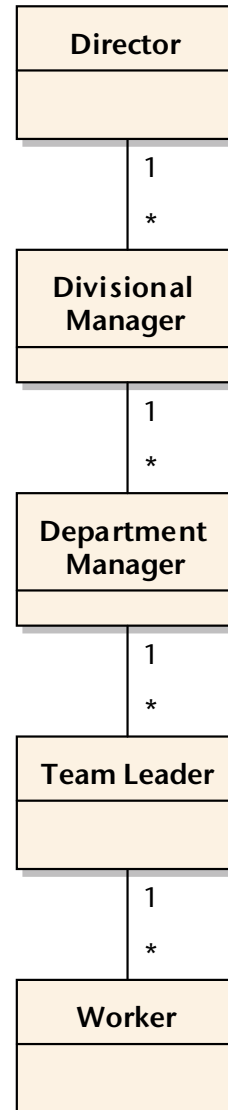
Business Patterns—People (cont.)

□ Separation of people from their positions:

- Dynamic specification of positions and their assignments to persons.

- Several superiority relationships were reduced to one relationship.

```
┌─────────────────┐
│    Position     │
├─────────────────┤
│                 │
│                 │
└─────────────────┘
         │ *
         │
         │ *
┌─────────────────┐  +superordinate
│     Person      │  0..1
├─────────────────┤────────────┐
│                 │            │
│                 │  +subordinate
└─────────────────┘  *
```
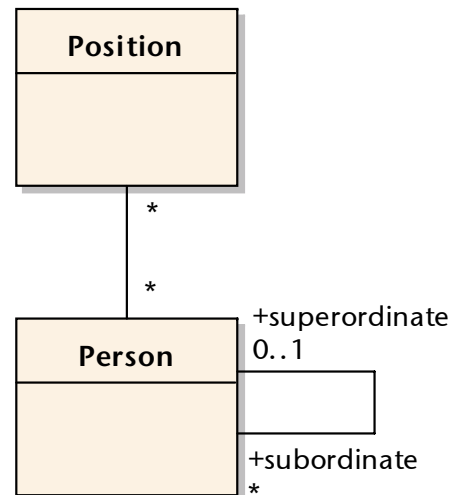
Business Patterns—People (cont.)

❑ Abstraction of various work relationships:

 • The possibility to express several different relationship types (including the original superiority relationship) dynamically.
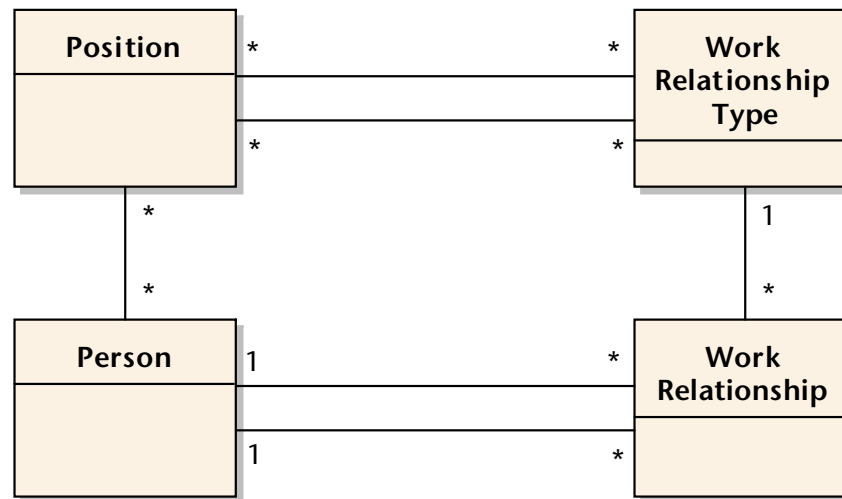
Business Patterns—People (cont.)

❒ Direct representation of an organization structure:

Business Patterns–People (cont.)

□ Flexible representation of an organization structure and its connection to positions:

- Application of the type abstraction and the relationship abstraction "meta" patterns.

☐   Direct representation of an address book:
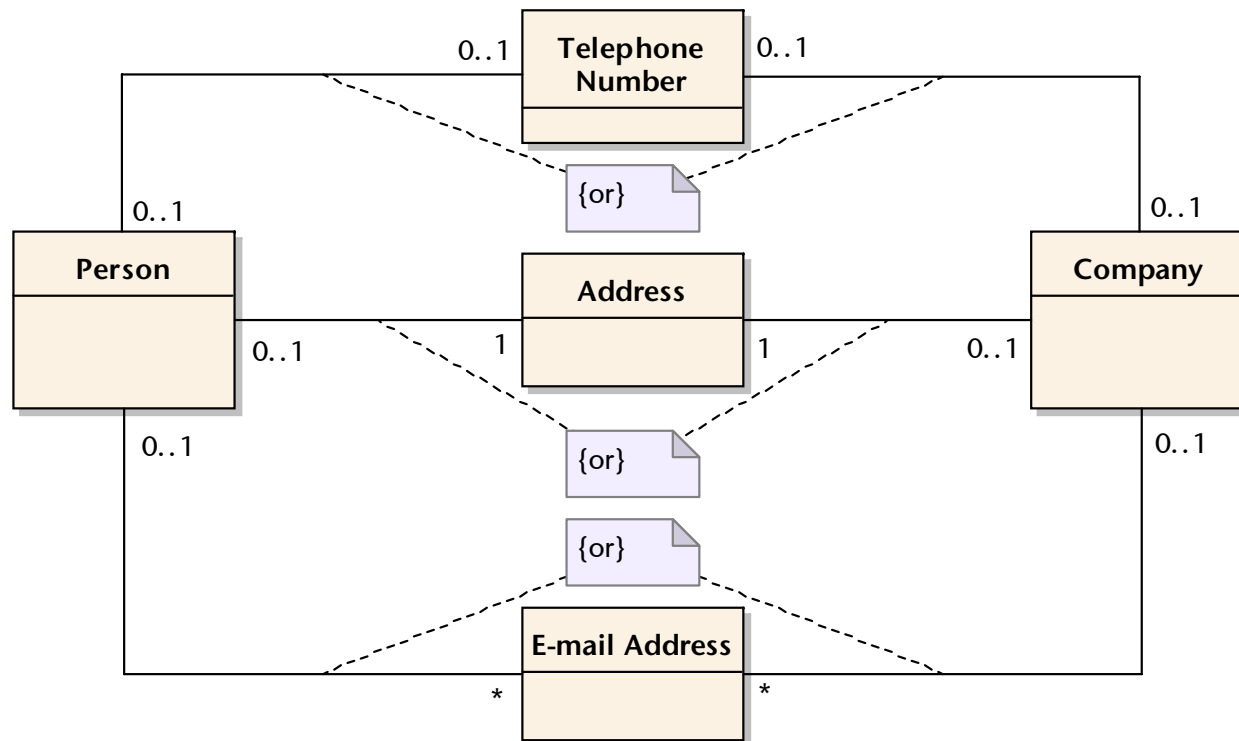
Business Patterns—People (cont.)

❏ Optimized version of an address book:

- The *Party* entity is a general representation of the *Person* and *Company* entities.

- Reduced amount of associations.

- No need for explicit constraints.
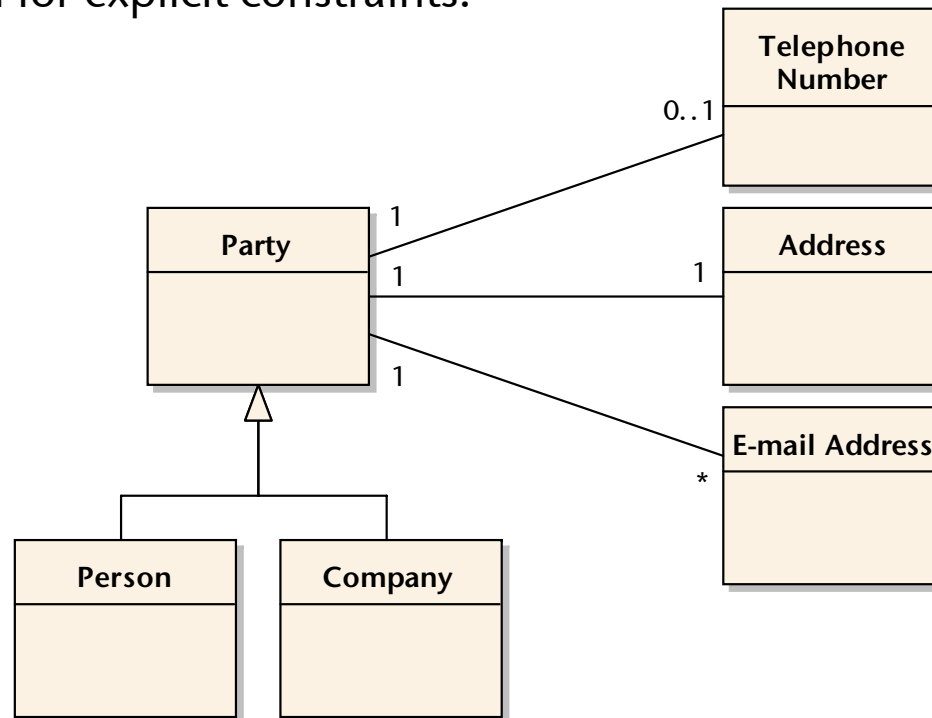
Business Patterns–People (cont.)

☐ Flexible representation of parties and their types:

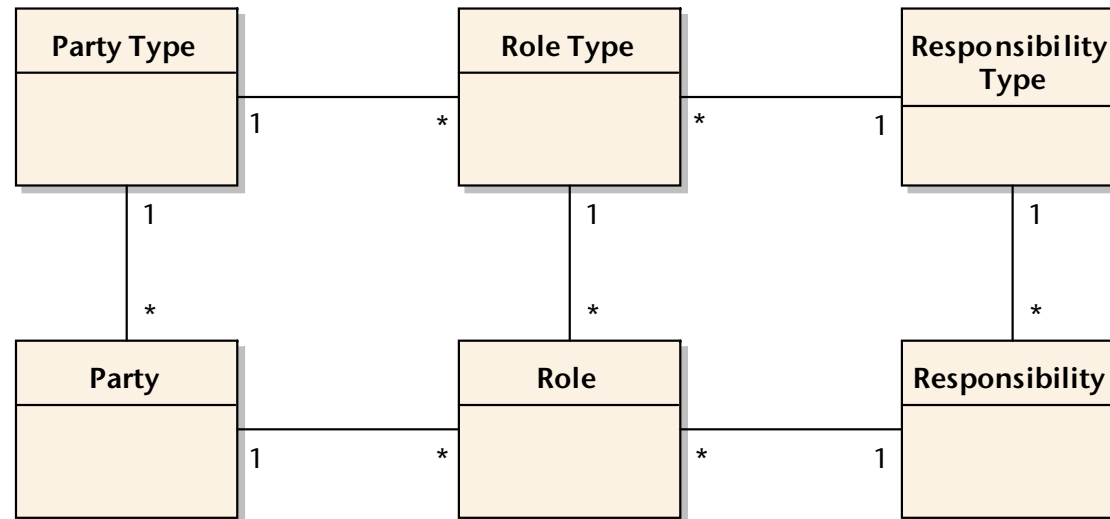● Application of the type abstraction "meta" pattern.

❑ Abstraction of the work relationship and organizational relationship by the generic *Responsibility* relationship:

- *Work Relationship* (*Type*) and *Organization Relationship* (*Type*) are represented by *Responsibility* (*Type*).

- Reduced amount of entities and relationships.

- Constraint: a responsibility can comprise just two parties.
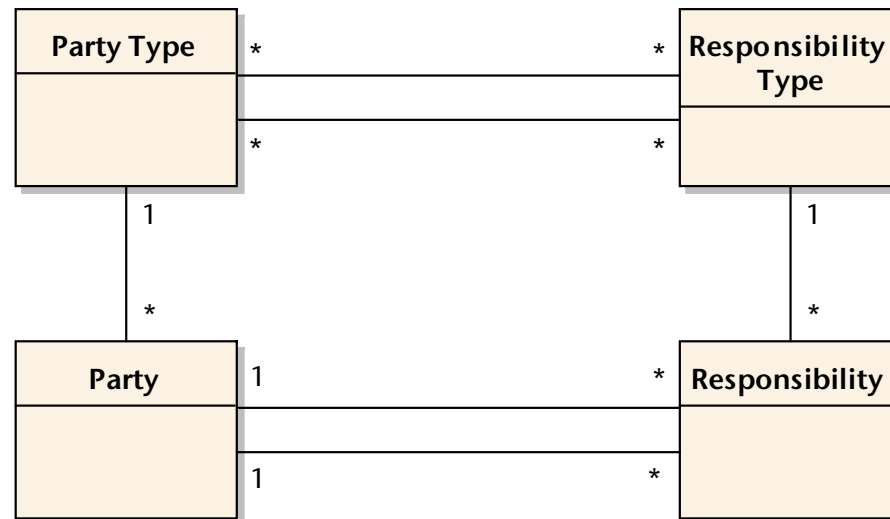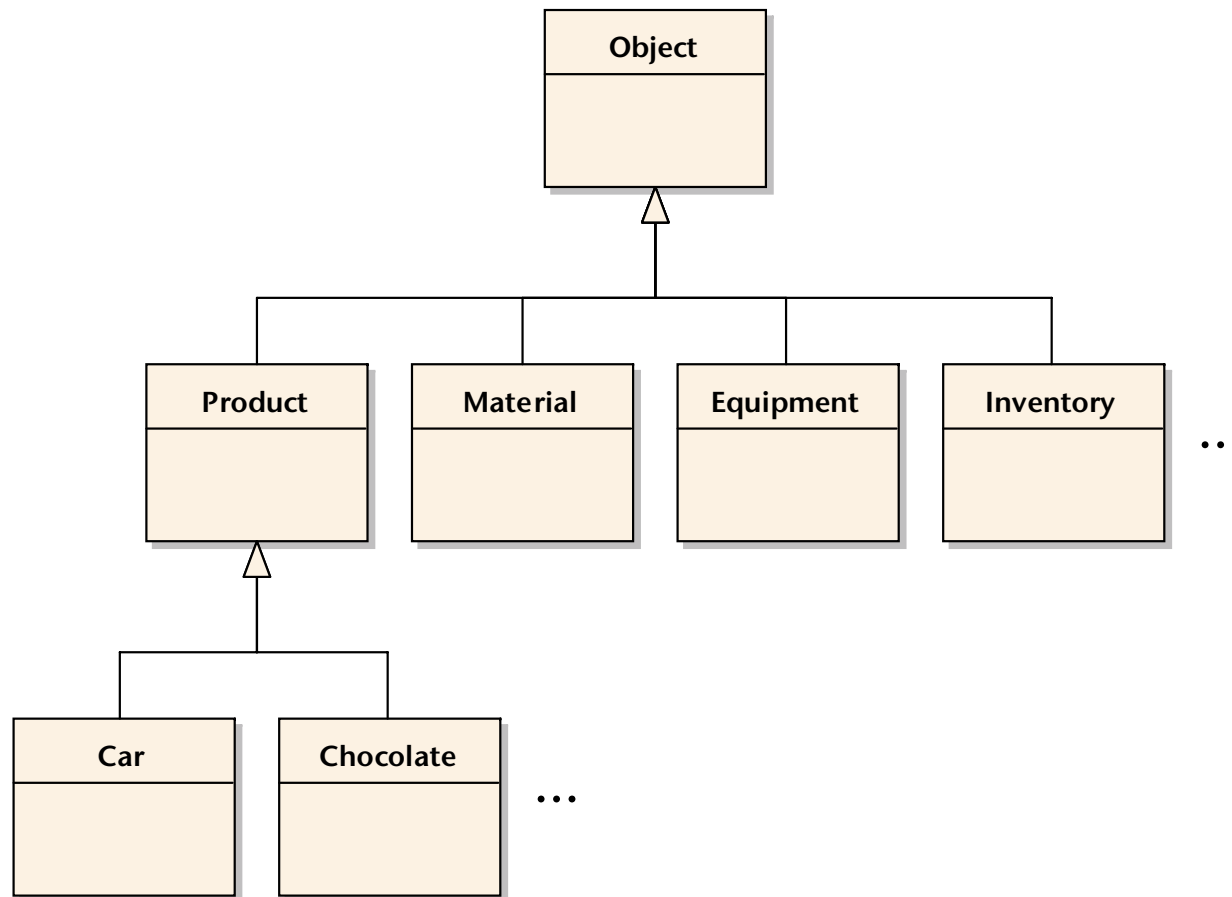
Business Patterns–People (cont.)

□ Extending the responsibility by several parties playing particular *roles*:

- Each responsibility can have several roles, each played by a (possibly different) party.
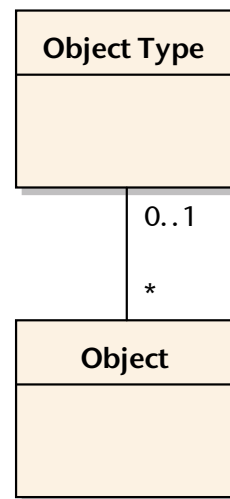
- The came at the type level.

Business Patterns–Objects

☐ Direct representation of business objects:

Business Patterns–Objects (cont.)

□ Flexible representation of objects and their types:

- Application of the type abstraction "meta" pattern.

```
┌─────────────────┐
│   Object Type   │
├─────────────────┤
│                 │
│                 │
└─────────────────┘
         │ 0..1
         │
         │ *
┌─────────────────┐
│     Object      │
├─────────────────┤
│                 │
│                 │
└─────────────────┘
```

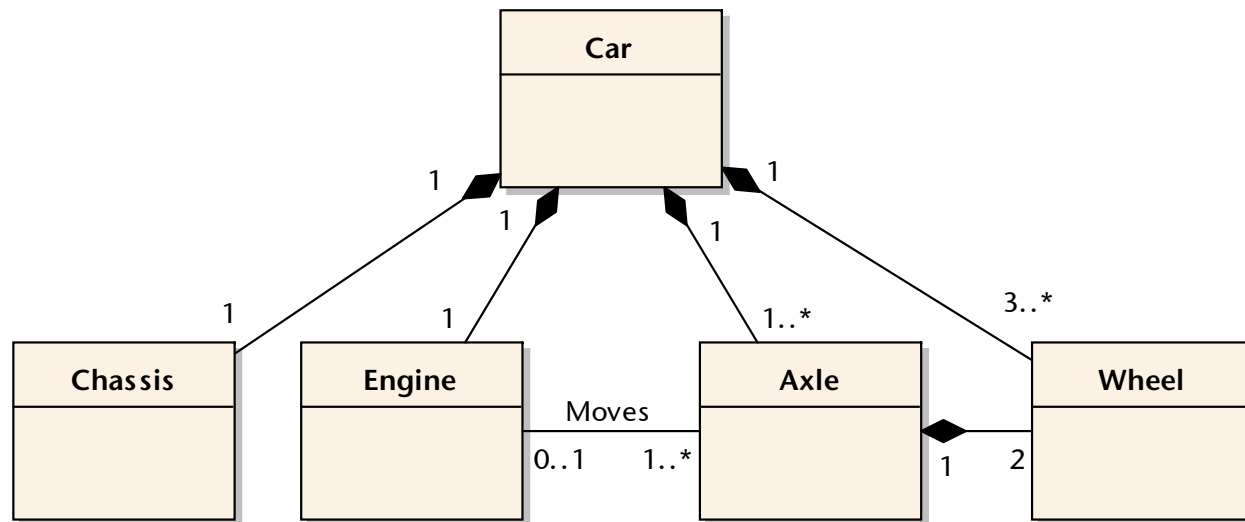# Business Patterns—Objects (cont.)

❑ Categorization of objects and their types:

- For instance, the object «Nissan Patrol» is of category «off-road».
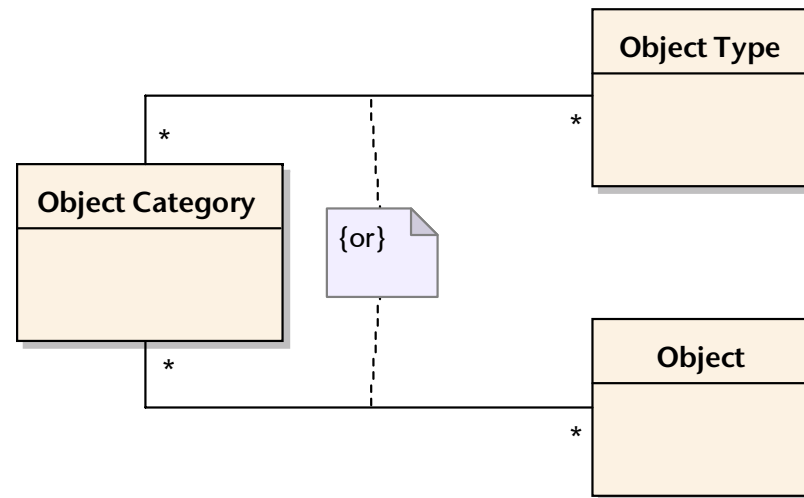
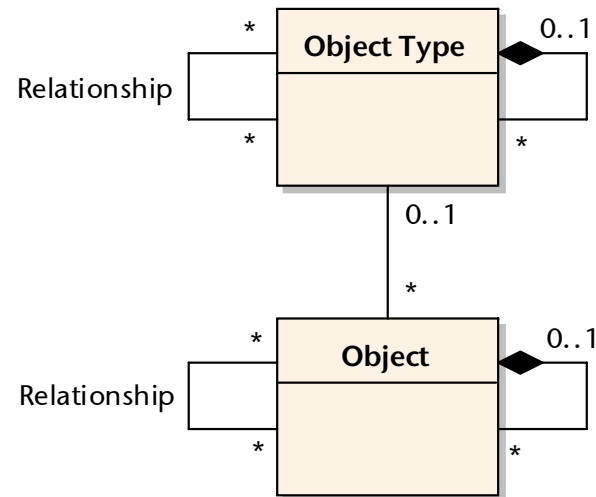# Business Patterns—Objects (cont.)

❑ Example of the direct representation of object structures (compositions and other relationships):
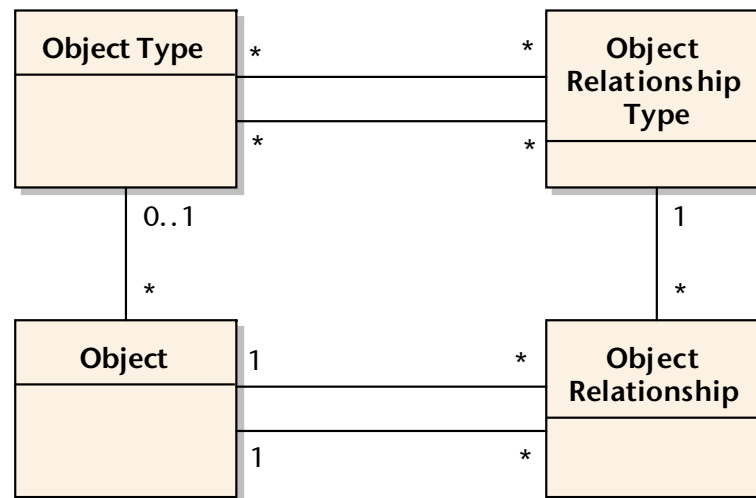
Business Patterns–Objects (cont.)

❏ Abstraction of object/object type composition and their other relationships:

- Relationships expressed directly by associations of entities.
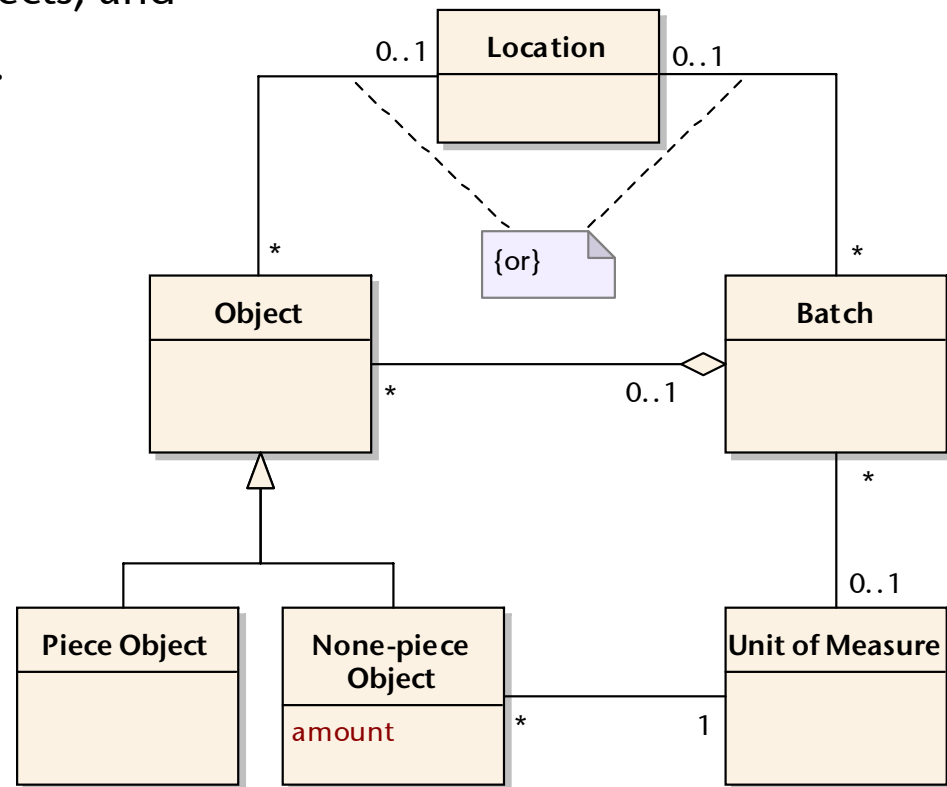
Business Patterns—Objects (cont.)

❒ Abstraction of object composition and other object relationships into a dynamic relationship:

- Application of the relationship abstraction "meta" pattern.

Business Patterns–Objects (cont.)

❑ Other aspects of business objects:

- placing at a location,
- batches (groups) of objects,
- piece and non-piece objects, and
- measurement of objects.

Application of Object Patterns in a Road Net